1994-09

Computer graphics tools for visualizing gravity gradient torques on a rigid spacecraft

Stewart, Jeffrey Alan

Monterey, California. Naval Postgraduate School

(1)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPUTER GRAPHICS TOOLS FOR VISUALIZING GRAVITY GRADIENT TORQUES ON A RIGID SPACECRAFT

by

Jeffrey Alan Stewart

September 1994

Thesis Advisor: I. Michael Ross

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704

1 AGENCY USE ONLY (Leave blank)	2 REPORT DATE September 1994	3 REPORT TYPE AND DATES COVERED Master's Thesis

4 TITLE AND SUBTITLE COMPUTER GRAPHICS TOOLS FOR VISUALIZING GRAVITY GRADIENT TORQUES ON A RIGID SPACECRAFT (U)	5 FUNDING NUMBERS
6 AUTHOR(S) Stewart, Jeffrey A	
7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8 PERFORMING ORGANIZATION REPORT NUMBER
9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10 SPONSORING/MONITORING AGENCY REPORT NUMBER

11 SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b DISTRIBUTION CODE A

13 ABSTRACT *(maximum 200 words)* This thesis provides students with a set of graphics tools allowing them to better visualize the effects of gravity-gradient torques on a rigid spacecraft in a low earth orbit. It allows the user to select from a variety of rigid bodies of different configurations, place them in any orientation at any altitude, apply the appropriate gravity-gradient moments to the body and immediately see the effect on the rigid body. This is accomplished through interactive computer graphics written to run on Silicon Graphics computers. The thesis includes a presentation of the theory involved in the programming of the physical properties and the discusses the basics of computer graphics including a more detailed look at the specific implementation for this thesis. A detailed user's guide is included to train students to use the tools as expeditiously as possible. It concludes with recommendations for further study in this area.

14. SUBJECT TERMS Gravity gradient torques, rigid spacecraft dynamics, computer graphics, visualization, PANSAT	15. NUMBER OF PAGES 143
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release, distribution is unlimited

COMPUTER GRAPHICS TOOLS FOR VISUALIZING GRAVITY GRADIENT TORQUES ON A RIGID SPACECRAFT

Jeffrey A. Stewart
Lieutenant, United States Navy
B.S., Tennessee Technological University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ASTRONAUTICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Author: [redacted]
Jeffrey A. Stewart

Approved by: [redacted]
I. Michael Ross, Thesis Advisor

[redacted]
Robert B. McGhee, Second Reader

[redacted]
Daniel J. Collins, Chairman
Department of Aeronautics and Astronautics

ABSTRACT

This thesis provides students with a set of graphics tools allowing them to better visualize the effects of gravity-gradient torques on a rigid spacecraft in a low earth orbit. It allows the user to select from a variety of rigid bodies of different configurations, place them in any orientation at any altitude, apply the appropriate gravity-gradient moments to the body and immediately see the effect on the rigid body. This is accomplished through interactive computer graphics routines, written to run on Silicon Graphics computers. The thesis includes a presentation of the theory involved in the programming of the physical properties and then discusses the basics of computer graphics including a more detailed look at the specific implementation for this thesis. A detailed user's guide is included to train students to use the tools as expeditiously as possible. It concludes with recommendations for further study in this area.

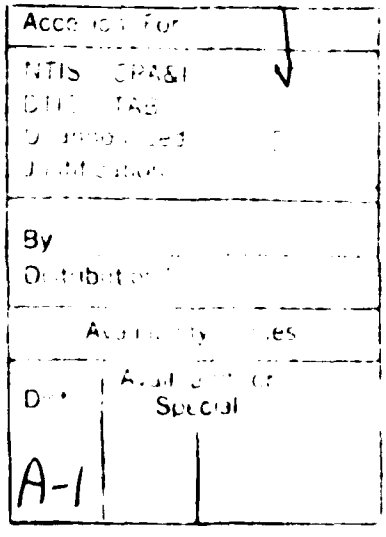

TABLE OF CONTENTS

I. INTRODUCTION AND BACKGROUND

A. INTRODUCTION

The goal of this thesis is to provide a tool for the visualization of the effects of gravity-gradient disturbance torques on a rigid-body spacecraft in an orbit around the earth. This is accomplished through the use of three-dimensional computer graphics written to emulate the laws of physics and the torques encountered by a spacecraft in a typical earth orbit. The system is fully interactive and allows the user to study spacecraft bodies of various shapes and sizes, including the Naval Postgraduate School's Petite Amateur Navy Satellite (PANSAT).

This thesis begins with a background discussion of the need for visualization tools of this sort. It then goes in to a discussion of the physics of gravity-gradient disturbance torques as well as the development of the equations for the gravity-gradient moment. Next it covers the graphical implementation of the program. A discussion of the results follows, including an example of the display screen and a tutorial for the user. Finally, a chapter on conclusions and recommendations is included.

B. BACKGROUND

This thesis was written to provide an educational tool for the analysis of spacecraft motion under the influence of the earth's gravity. Presently, there exists several computer software programs that can be programmed to simulate the dynamics of a rigid body [Ref. 1]. There are some that are programmed to simulate spacecraft dynamics and control [Ref. 2]. There are routines that enable the user to study spacecraft orbits and ground

tracks [Ref. 3]. All of the above routines have one or more disadvantages. Some are difficult to work with and lack a graphical user's interface [Ref. 2]. Others give only orbital parameters and tell nothing of what is physically happening to the spacecraft [Ref. 3]. Some display the spacecraft in a wireframe representation [Ref. 3]. Still others provide analysis in the form of graphs and plots of data [Ref. 1]. After using some of the routines currently available, it became obvious that this was not the optimum way to determine what is actually happening to a spacecraft in orbit. Plots of data are helpful and can yield useful information, but usually only after in-depth study. A more intuitive method that would allow a user to see the spacecraft in motion on the display is needed. The software described by this thesis was written in an attempt to overcome the aforementioned disadvantages and provide that intuitive approach. This program will allow the user to view a spacecraft-body under the influence of gravity-gradient torques. One can describe the spacecraft body as a block, sphere, cylinder or create a new configuration such as PANSAT. The user will be able to specify the body's mass, size and dimension, as well as place it at any altitude in any initial orientation. One can then apply the initial orbital rotations and the gravity-gradient torques that would be acting on the spacecraft. This tool allows the user to experiment with different situations and learn from the effects of those situations. There is no better way to learn than by a "hands on" approach and this thesis will allow the user to sit at a computer screen and "play" with different values and see immediately the effect of those values on his spacecraft.

II. DYNAMICS

A. INTRODUCTION

This chapter discusses the theory on the dynamics demonstrated in this thesis. It begins with a discussion of frames of reference, followed by an explanation of Euler's equations of motion and finishes with the gravity-gradient moment equation. For the sake of brevity, the equations used will not be derived here. For a derivation of the stated equations, see [Ref. 4,pp. 106-112] and [Ref. 5,p. 113].

B. FRAMES OF REFERENCE

The motion of a rigid body can be described in several different ways, depending upon the frame of reference used. As a result, the description of the body's motion is not complete without also describing the frame of reference. The two frames of reference used in this thesis are the orbit reference frame and the body reference frame. The orbit frame consists of three orthogonal axes, $\mathbf{o}_1$, $\mathbf{o}_2$ and $\mathbf{o}_3$, that allows one to describe the motion of a spacecraft with respect to its orbit plane. For the purposes of this thesis, $\mathbf{o}_1$ will be anti-earth pointing, $\mathbf{o}_2$ will be in the direction of flight and $\mathbf{o}_3$ will be in the direction of the orbit normal (see Figure 2.1). The body frame is fixed to the spacecraft's principal axes (see Figure 2.2) and coincides with the orbit frame in the absence of spacecraft roll, pitch or yaw. If there is a roll, pitch or yaw, these two frames are separated by a rotation through that angle about the appropriate axis (see Figure 2.3). One can describe the spacecraft's motion in either frame and can switch back and forth between frames by using

a Direction Cosine Matrix (DCM) to convert from one set of coordinates to the other. A complete discussion of DCM's can be found in [Ref. 6].

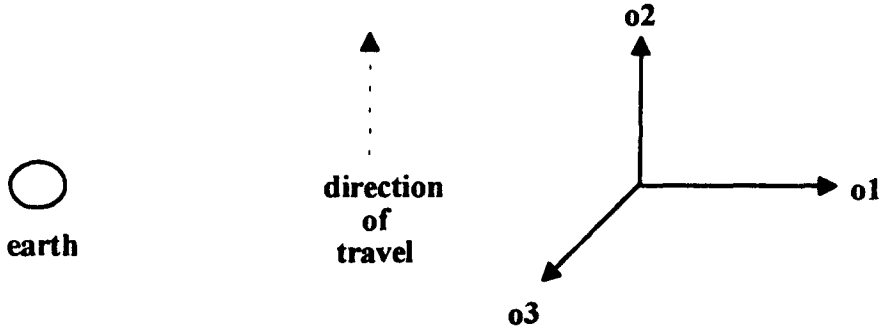

Figure 2.1. Orbit Reference Frame

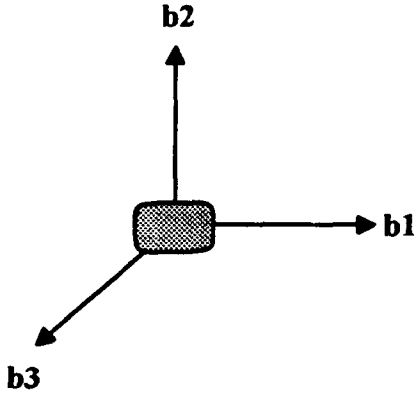

Figure 2.2. Body Reference Frame

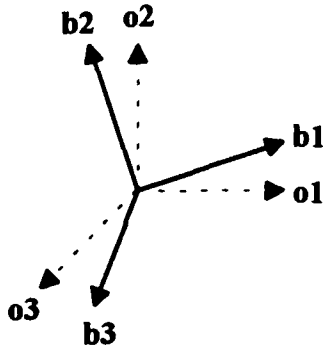

Figure 2.3. Orbit and Body Frames

C. EULER'S EQUATIONS OF MOTION

Euler's equations of motion are a set of three coupled differential equations that describe the effect of applied moments on a rigid body. In the case where the products of inertia are zero (principal axes of moment of inertia), Euler's equations of motion are [Ref 4.p. 112]

$$M_x = I_{xx}\dot{\omega}_x + \omega_y\omega_z (I_{zz} - I_{yy}) \tag{3.1}$$

$$M_y = I_{yy}\dot{\omega}_y + \omega_x\omega_z (I_{xx} - I_{zz}) \tag{3.2}$$

$$M_z = I_{zz}\dot{\omega}_z + \omega_x\omega_y (I_{yy} - I_{xx}) \tag{3.3}$$

or written as the dynamical equations of motion for a rigid body

$$\dot{\omega}_x = \frac{M_x}{I_{xx}} + \omega_y\omega_z\left(\frac{I_{yy} - I_{zz}}{I_{xx}}\right) \tag{3.4}$$

$$\dot{\omega}_y = \frac{M_y}{I_{yy}} + \omega_x\omega_z\left(\frac{I_{zz} - I_{xx}}{I_{yy}}\right) \tag{3.5}$$

$$\dot{\omega}_z = \frac{M_z}{I_{zz}} + \omega_x\omega_y\left(\frac{I_{xx} - I_{yy}}{I_{zz}}\right) \tag{3.6}$$

where

$\dot{\omega}_x$, $\dot{\omega}_y$, and $\dot{\omega}_z$ are the rates of change of the angular velocity components about the x, y, and z axis, respectively

ω_x, ω_y, and ω_z are the angular velocity components about the x, y, and z axis, respectively

M_x, M_y, and M_z are the moments about the x, y, and z axis, respectively

I_{xx}, I_{yy}, and I_{zz} are the principal moments of inertia about the x, y, and z axis, respectively.

Furthermore, $I_{xx} = \int_m (y^2 + z^2)\, dm$, (3.7)

$I_{yy} = \int_m (x^2 + z^2)\, dm$, and (3.8)

$I_{zz} = \int_m (x^2 + y^2)\, dm$ (3.9)

with dm being a particle of differential mass and x, y, and z being the distance from the center of mass to that mass particle. It should be noted that M is the sum of all external moments on the spacecraft such as gravity gradient moment, solar pressure moment and control moment, but for the purpose of this thesis, we will only consider the gravity gradient moment.

D. GRAVITY-GRADIENT TORQUES

A gravity-gradient torque applied to a spacecraft body is due to the difference in the distances between the various mass points on the spacecraft body and the center of mass of the earth. The magnitude of this torque is dependent upon many factors. These factors include the principal moments of inertia, which take into account the moment arm of the point mass measured from the center of mass, the altitude of the spacecraft, and the orientation of the spacecraft with respect to its orbit frame. The gravity gradient torque on an arbitrary spacecraft can be approximated by [Ref. 5,p. 113]

$$\boldsymbol{M}_G = \frac{3\mu_e}{R^3} \begin{Bmatrix} (I_{zz} - I_{yy})C_{12}C_{13}\,\mathbf{x} \\ (I_{xx} - I_{zz})C_{11}C_{13}\,\mathbf{y} \\ (I_{yy} - I_{xx})C_{12}C_{11}\,\mathbf{z} \end{Bmatrix} \tag{3.10}$$

where

μ_e is the earth's gravitational constant and equals 398601.395 km^3/sec^2

R is the orbit radius and equals the earth's radius (6378.14 km) plus the orbit altitude

C_{ij}'s are the direction cosine matrix (DCM) elements used to express body coordinates in the orbit frame, where $C_{ij} = \mathbf{o}_i \cdot \mathbf{b}_j$

Substituting the elements of Eq 3.10 into Eqs 3.4, 3.5, and 3.6 the dynamical equations of motion including gravity gradient torques become

$$\dot{\omega}_x = \left(\frac{3\mu_e}{R^3}(I_{zz} - I_{yy})C_{12}C_{13} - \omega_y\omega_z(I_{zz} - I_{yy}) \right) / I_{xx} \tag{3.11}$$

$$\dot{\omega}_y = \left(\frac{3\mu_e}{R^3}(I_{xx} - I_{zz})C_{11}C_{13} - \omega_z\omega_x(I_{xx} - I_{zz}) \right) / I_{yy} \tag{3.12}$$

$$\dot{\omega}_z = \left(\frac{3\mu_e}{R^3}(I_{yy} - I_{xx})C_{12}C_{11} - \omega_x\omega_y(I_{yy} - I_{xx}) \right) / I_{zz} \tag{3.13}$$

These equations of motion that take into account the gravity gradient torque are used in the simulation.

III. GRAPHICS AND IMPLEMENTATION

A. INTRODUCTION

This chapter will discuss how the physics covered in the previous chapter is converted from an idea and equations on paper to computerized graphics on the screen. It will begin with a brief primer on the basics of computer graphics and end with the specifics of the implementation of the gravity gradient problem.

B. COMPUTER GRAPHICS

The programs contained in this thesis were developed and tested on a Silicon Graphics Élan computer. Listed below are the minimum hardware and software requirements for proper execution of the code.

Hardware

- 1 50 MHz IP20 Processor
- FPU: MIPS R4010 Floating Point Processor Chip
- CPU: MIPS R4000 Processor Chip
- Data Cache: 8 KB
- Instruction Cache: 8 KB
- Secondary Cache: 1 MB
- Main Memory: 64 MB
- IRIS Audio Processor: Revision 10
- Graphics Board: GR2-Élan
- Mouse

Software

- Operating System Silicon Graphics Irix version 4.05
- Compiler Silicon Graphics C++ Compiler version 3.0

It is important to note that a graphics board capable of Z Buffering is essential to the correct view of the graphics screens. The software will still function without it but the displays will be grainy and some objects may appear incomplete. It should also be noted that the software will run on any operating system version after 4.05.

The world of computer graphics is complex and a full discussion of the subject is beyond the scope of this thesis. For a discussion of the basics of computer graphics and how they are implemented on Silicon Graphics computers and used by this thesis, see Haynes [Ref. 7,pp. 16-23].

C. IMPLEMENTATION

This thesis was conceived as an extension of Haynes [Ref. 7]. As such it draws heavily upon the groundwork previously developed. The goal is to start with a general routine and develop it into a specific application to be used by students in their study of spacecraft attitude dynamics. All coding is done in the programming language C++.

The software discussed in this thesis is written using several basic tools, some of which were previously developed by Haynes. The C++ programming language supports a data structure called a *class* and this data structure is used extensively to define the various objects used in the thesis. A *class* data structure is such that it contains the information that defines the data structure as well as all the functions that can operate on

the data structure. The advantage to this is self-contained, re-usable code is that it is resistant to data corruption by routines that are not intended to have access to the data in question. This thesis it built around three basic classes: the vector3D class, the matrix3x3 class and the rigid_body class. The vector3D class contains a three dimensional vector data type used to store such information as a position vector or velocity vector. Included in this class are several functions used to perform various operations on the vector such as vector arithmetic and normalization. The matrix3x3 class contains a data type representing a 3x3 matrix used to store such information as rotation matrices between two reference frames. Also included in this class are functions used to perform matrix operations such as matrix algebra. Finally the rigid_body class contains a data type used to represent any rigid body and contains information such as mass, size, location, velocity, acceleration or moments of inertia of the rigid body. Most of this information is contained in either the vector3D or the matrix3x3 class within the rigid_body class. For example, velocity is stored in a vector3D class which is in turn stored in the rigid_body class and the inertia matrix is stored in a matrix3x3 class which is stored in the rigid_body class. The class also contains functions that define the various operations that can be performed on the rigid body, such as assigning it a velocity or position, rotating it or changing its size or shape. For a detailed explanation of these classes and their associated functions, see Haynes.

Another fundamental building block of this thesis is the graphics functions used to drive the display. The graphics functions are used to build and display the background

screen as well as the objects displayed on the background They consist of routines to ready the screen for display, routines to change the position from which you are looking and the position to which you are looking, routines to display various data fields on the background and routines to display various objects on the screen such as rigid bodies or more specifically, spacecraft. These displayable objects are graphics models that are built outside of the software in this thesis and then read into memory for display. The construction of these models is quite complex and will not be discussed in this thesis. For a detailed discussion of these graphics models, see Zyda [Ref. 8].

Along with the graphics models themselves, a method of providing motion to them is needed. Providing motion for the displayed objects is accomplished by the use of numerical integrators such as the Runga-Kutta Fourth Order Method and the Runga-Kutta Adaptive Step Method (the latter being also know as the Runga-Kutta Fourth/Fifth Order Method). These integrators take the current state of the object and the moment applied to it to determine the next state.

Finally, a routine is needed to collect and drive all the aforementioned building blocks. This is accomplished through a "main" program that determines what action needs to take place, when it needs to take place and calls the appropriate routine to make it happen. This routine is the "brains" of the program and it uses the classes, graphics, and integrators as merely tools to accomplish its mission.

IV. USER'S GUIDE

A. INTRODUCTION

A software package is useless without the training to use it. This chapter will provide the user with that training. It will present the user with a step-by-step discussion of how to use the gravity-gradient visualizer software. It contains the visualizer display screens including the control buttons that allow the user full control over the simulation. It also discusses the different procedures required depending on the type of rigid body with which the user wishes to work.

B. TUTORIAL

After logging on the Silicon Graphics Computer, start the gravity-gradient visualizer software by typing **ggrad** at the UNIX prompt. It will then take 5-10 seconds for the software to load and for the initial graphics screen to be displayed. This initial screen will consist of a control window and a main display window. The initial control window and the initial main display window will appear as in Figure 4.1. The initial values for inertia, mass, angular velocity and angular momentum are displayed in the main display window in addition to the initial rigid body All functions are accessed by clicking on the appropriate field once with the left mouse button. It is important to note that mouse clicks are valid only if they are performed with the mouse pointer in the control window. Clicking in the main display window will have no effect. The only exception to this is that the right mouse button can be held down anywhere on the screen to make a selection to exit the program. The orientation of the main display screen is as follows:

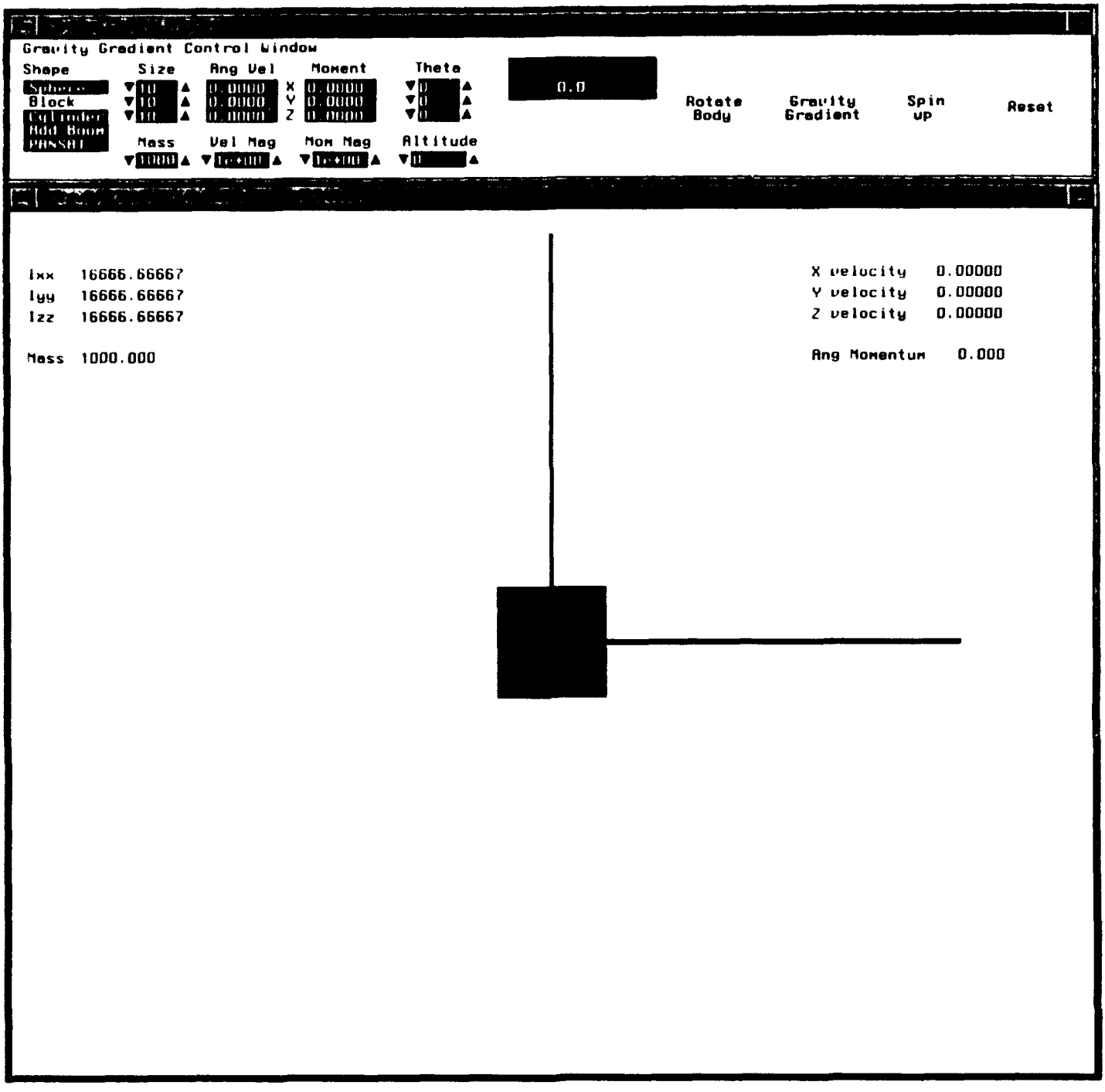

Figure 4.1. Display with "Block" Selected.

the red axis is anti-earth pointing (o_1), the blue axis is along the direction of motion (o_2) and the black axis is along the orbit normal (o_3). As such, the user is looking down on the orbit plane from above with the earth off the left side of the screen (see Figure 2.1).

A number of options are available to the user when the initial screen is displayed and the selection of these options will be reflected in the displayed rigid body. First, the user can select the shape of the object he wishes to display, choosing from a sphere, cube or cylinder with or without a gravity-gradient boom attached or a model of the PANSAT (see Figure 4.1). To make this selection, the user should click on the word for the appropriate shape. The procedures for the sphere, cube and cylinder are different than those for a sphere, cube and cylinder with a boom, which are different from those for PANSAT. These three cases will be discussed separately, in a step-by-step fashion.

If the user selects a sphere, cube or cylinder, the procedure would be as follows:

- Change the mass of the object by clicking the up or down arrows either side of the mass field. This will increase or decrease the mass by 50 kilograms for each click. Changing the mass will result in the moments of inertia being re-calculated and redisplayed.
- Change the size of the object by clicking on the up or down arrows either side of the size field. This will increase or decrease the size by one meter for each click. The size of the selected object can be changed along any axis and changing the size will result in the moments of inertia being recalculated and re-displayed.
- Select a rotation angle which represents an initial error in orientation. This is

accomplished by clicking the arrows either side of the field marked "Theta", increasing or decreasing the angle by five degrees for each click.

- Select an altitude for the rigid body by clicking the arrows either side of the altitude field, increasing or decreasing the altitude by 500 kilometers for each click. This will cause an initial angular velocity about the **z** axis to be calculated and displayed in the field marked "Ang Vel".
- Click on the button labeled "Rotate Body" to affect the rotation by the angle selected above in the field marked "Theta".
- Click on the button labeled "Spin Up" to start the rotation about the **z** axis with the angular velocity displayed in the "Ang Vel" field.
- Click on the button labeled "Gravity Gradient" to affect the gravity gradient moment on the rigid body. This will result in the torques being calculated from the altitude and initial orientation and those being displayed in the field marked "Moment".
- To terminate the simulation and reset the initial values, click on the button labeled "Reset".

At first, the displayed body may not appear to be moving. It is, but very slowly. This routine was built to display real-time spacecraft motion. As such, the user views the real-time angular motion of the spacecraft. To speed up this displayed motion, an option was added to allow the user to increase or decrease the magnitude of these values by a factor of ten. This is accomplished with the arrows either side of the fields marked "Vel

Mag" and "Mom Mag". Thus, the user can increase the velocity magnitude and the moment magnitude in order to better visualize their effects on the spacecraft.

If the user selects a sphere, cube or cylinder with the gravity gradient boom, the procedure would be changed slightly. The control window and the main display window will now appear as in Figure 4.2, assuming that the cube with a boom is selected. The changes in the procedure are due to the added complexity of calculating the moments of inertia for the new rigid body. As a result, if the user changes the mass, the values are changed but not the moments of inertia. Additionally, the size field is now changed to the inertia field. The displayed inertia values are for a standard body of 1000 kilograms, a size of one meter in the **x**, **y** and **z** directions and a six meter massless boom with a two kilogram tip mass. Therefore, instead of changing the mass and size, and then re-computing the inertia, the user is able to change the inertia directly. This is accomplished by clicking the arrows either side of the "Inertia" field to increment or decrement the inertia by five kilogram-meter^2 per click. The remainder of the procedure is the same as for the boomless cases.

The final set of procedures involves PANSAT. The procedures are similar to those for the rigid bodies with a boom with some minor exceptions. The similarities are once again due to the complexity in calculating the moments of inertia of PANSAT. The differences are due to the need for much greater accuracy in the inertias and the much smaller initial attitude errors expected. As a result, the control window and the main display window will appear as in Figure 4.3. Note that the changes in the "Theta" field

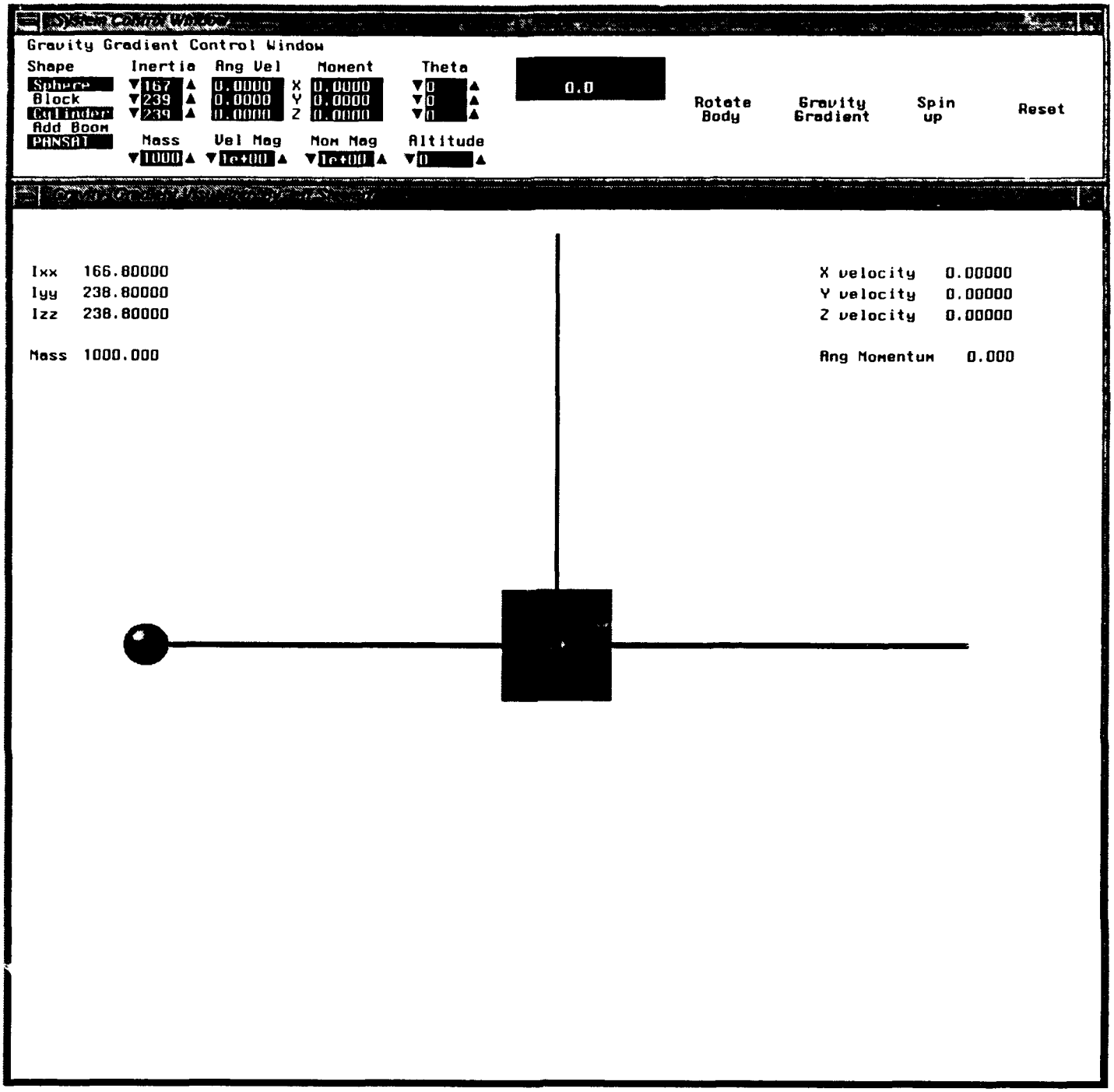

Figure 4.2. Display with "Block" and "Add Boom" Selected.

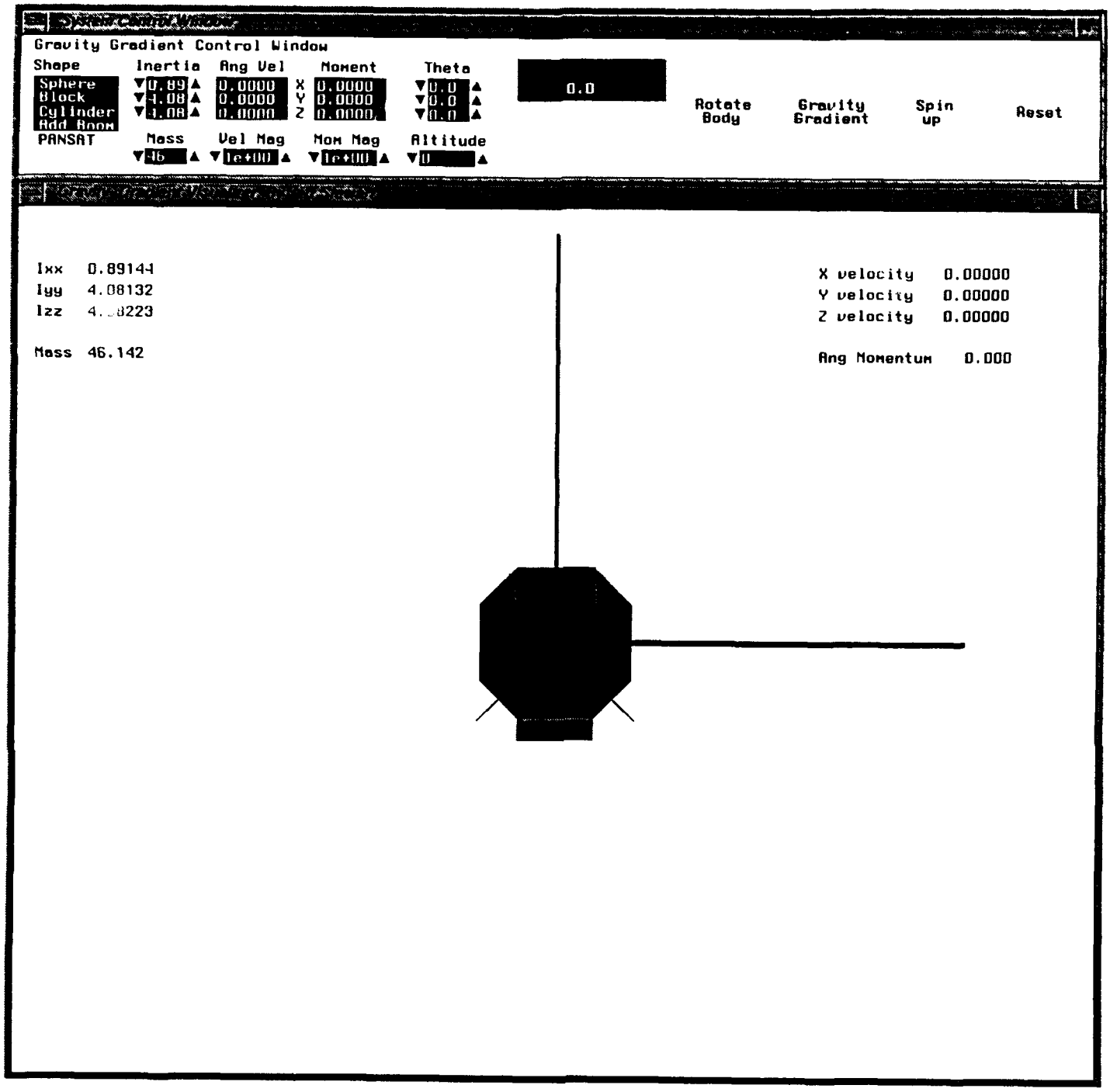

Figure 4.3. Display with "PANSAT" Selected.

will now be in tenths of a degree increments vice five degree increments and the changes in the inertia data field will be incremented by five one-hundredths of kilogram-meter^2 instead of one kilogram-meter^2. In addition to these differences, the changes in altitude will be in twenty-five kilometer increments vice 500 kilometer increments. These smaller increments will allow the user to make finer adjustments to more closely represent the actual PANSAT mission specifics. All other procedures are the same as for the case of the boomless rigid bodies.

In-depth testing has gone into the above procedures with a goal of being user friendly in mind. Unfortunately, this goal was at times compromised slightly in order maintain a user interface consistent with that defined by Haynes [Ref. 7] and to accomplish the greater goal of providing an easy to use tool for the desired analysis. With the extensive software features and this detailed user's guide usablility will not be a problem.

As a final note, every attempt was made to arrive at values for the data fields that are applicable to as many situations as possible. It is recognized that in some cases the chosen values for the data fields are not entirely appropriate. It is not recommended that the user alter this application in an attempt to tailor it to his specific needs. However, if one possesses the requisite knowledge of graphics and programming in C++ to perform this a task, such changes are possible. Appendix I contains suggestions to aid in altering the preset data fields.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The goal of this thesis was to provide a tool for students to use in visualizing the effects of gravity-gradient disturbance torques on a rigid-body spacecraft in a typical low earth orbit. After simulating this same problem with software products that produced data plots as results, it is obvious that this thesis provides a better method of analysis. A late, but very useful addition to the thesis was the implementation of PANSAT. The Space Systems Academic Group (SSAG), charged with oversight of PANSAT, was very interested in the final results. The designers of PANSAT were in need of a tool that would allow them to learn how PANSAT would behave after release from the Space Shuttle. They also needed a way to determine how the solar cells on PANSAT would be shadowed in various attitudes. This thesis provides them with a tool for this analysis. This was an unexpected benefit of the thesis and at the same time validated that 1) this type of software was needed and 2) this thesis would have future real-world applications. In this respect, the goal was met and this thesis has been successful.

B. RECOMMENDATIONS

There are several areas for further study regarding this thesis. First, as mentioned earlier, the moment on a rigid body consists of a gravity-gradient moment, a solar pressure moment and a control moment. This thesis could be further developed to include the solar pressure moment and a control moment in the disturbance torques experienced by the spacecraft. In addition to the control torque, a means of damping the spacecraft control

could be added. Another area for further study could be a feature to allow the user to obtain a hard copy of selected data in order to reinforce the graphics display.

A very in-depth follow-on to this thesis would be to convert all the existing code from Graphics Library (GL) programming tools to MOTIF programming tools. Under GL all of the displays must be manually programmed while MOTIF handles the low-level programming chores such as constructing pulldown menus and data entry fields. This would enable the programmer to spend his time on programming the simulation vice programming the basic graphics functions and would add some very useful options such as the ability to arbitrarily specify data instead of relying on fixed increments of data fields. This would be a major undertaking, as this thesis consists of several thousands of lines of code.

APPENDIX A. GRAVITY GRADIENT VISUALIZER CODE

A. CONSTANTS HEADER FILE

```
#ifndef CONSTANTS_H
#define CONSTANTS_H
#include "vector3D.H"

const long double pi = 3.145926536;
const long double deg_rad = 0.0174532925;        //conversion from deg to rad
const long double mu_earth = 398601.395;         // earths gravitational constant
                                                 // (km^3/sec^2)
const long double earth_radius = 6378.14;        // earth's radius (km)

#endif
```

B. GRAVITY GRADIENT HEADER FILE

```
#define BASE_H
#include "constants.H"
#include "rigid_body.H"
#include "menu.H"
```

C. GRAVITY GRADIENT SOURCE FILE

```
#include "base.H"
#include <stdlib.h>
#include <iostream.h>

void main()
{
        int section = 0, bypass = 0, NO_GO = 0, mx = 0, my = 0, GO = 0, obj = 2,
        go_next = 0, axis1 = 0, axis2 = 1, axis3 = 2;

        matrix3x3 rotation, mat1, mat2, mat3;

        vector3D angular_velocity, inertia,size(10,10,10), theta(0,0,0),
                        pansat_size(10,10,10), boom_size(10,10,10), reuse,
                        omega_vec(0,0,0), initial_omega(0,0,0);

        double mass = 0.0, duration = 99999999.0, elapsed_time = 0.0, step = 0.0,
                am, am1, am2, am3, mag = 1.0, total_time = 0.0, vel_mag = 1,
                mom_mag = 1, rot = 0.0, altitude = 0.0, omega = 0.0, radius = 0.0,
```

```
        x = 0, y = 0, z = 0, mox = 0, moy = 0, moz = 0;

initialize();
initialize_menu();
init_control_window();
main_window();

rigid_body cube(1), ball(2), cylinder(3), pansat(60);
rigid_body ball_boom(71), cube_boom(72), cylinder_boom(73);
rigid_body frame(100), axis(200), reuse_body;
reuse_body.assign_shape(cube.return_shape());
reuse_body.assign_mass(cube.return_mass());
reuse_body.assign_size(size);
reuse_body.add_axis();
reuse_body.assign_type(1);
reuse_body.compute_inertia();
mass = reuse_body.return_mass();
set_target(0.0,0.0,0.0);
set_eye(0.0, 0.0,100);
set_time();

while (section != 99)                    // while exit not selected
{
    section = queue_test();              // what type of interrupt event
    set_delta();
    view();

    // draw the controls screen
    gyro_controls(x,y,z,obj,altitude,size,theta,mass,mox,moy,moz, vel_mag,
                  mom_mag, elapsed_time, inertia);

    if (bypass > 0 && bypass < 6)        // system delay for mouse input
                                         // timing
        bypass++;
    else
        bypass = 0;

    if (section > 99999)                 // if the event was a mouse
                                         // selection
    {
        mx = section / 100000;           // decode mouse x & y coords
        my = section - (mx * 100000);
```

```
if (!bypass)
{
    bypass = 1;

    if (obj == 7)                          // PANSAT
    {

        // Inertia Decrease
        if (mx > 113 && mx < 129)
        {
            if (my > 936 && my < 953)   // Ixx
                if (inertia[0] > 0.05)
                    inertia[0] = inertia[0] - 0.05;

            if (my > 923 && my < 937)   // Iyy
                if (inertia[1] > 0.05)
                    inertia[1] = inertia[1] - 0.05;

            if (my > 909 && my < 924)   // Izz
                if (inertia[2] > 0.05)
                    inertia[2] = inertia[2] - 0.05;

            reuse_body.assign_inertia(inertia);
        }

        // Inertia Increase
        if (mx > 165 && mx < 181)
        {
            if (my > 936 && my < 953)          // Ixx
                inertia[0] = inertia[0] + 0.05;

            if (my > 923 && my < 937)          // Iyy
                inertia[1] = inertia[1] + 0.05;

            if (my > 909 && my < 924)          // Izz
                inertia[2] = inertia[2] + 0.05;

            reuse_body.assign_inertia(inertia);
        }
    }

    else if ((obj > 3) && (obj < 7))        // boom object
    {
```

```
        // Inertia Decrease
        if (mx > 113 && mx < 129)
        {
            if (my > 936 && my < 953)          // Ixx
                if (inertia[0] > 1)
                    inertia[0] = inertia[0] - 5;

            if (my > 923 && my < 937)          // Iyy
                if (inertia[1] > 1)
                    inertia[1] = inertia[1] - 5;

            if (my > 909 && my < 924)          // Izz
                if (inertia[2] > 1)
                    inertia[2] = inertia[2] - 5;

            reuse_body.assign_inertia(inertia);
        }

        // Inertia Increase
        if (mx > 165 && mx < 181)
        {
            if (my > 936 && my < 953)          // Ixx
                inertia[0] = inertia[0] + 5;

            if (my > 923 && my < 937)          // Iyy
                inertia[1] = inertia[1] + 5;

            if (my > 909 && my < 924)          // Izz
                inertia[2] = inertia[2] + 5;

            reuse_body.assign_inertia(inertia);
        }
    }

    else                                       // cube, sphere, or cylinder
    {

        // Size Decrease
        if (mx > 113 && mx < 129)
        {
            if (my > 936 && my < 953)          // size along x axis
                if (size[0] > 1.0)
```

```
                size[0] = size[0] - 1;

        if (my > 923 && my < 937)          // size along y axis
            if (size[1] > 1.0)
                size[1] = size[1] - 1;

        if (my > 909 && my < 924)          // size along z axis
            if (size[2] > 1.0)
                size[2] = size[2] - 1;

        reuse_body.assign_size(size);
        reuse_body.compute_inertia();
    }

    // Size Increase
    if (mx > 165 && mx < 181)
    {
        if (my > 936 && my < 953)          // size along x axis
            size[0] = size[0] + 1;

        if (my > 923 && my < 937)          // size along y axis
            size[1] = size[1] + 1;

        if (my > 909 && my < 924)          // size along z axis
            size[2] = size[2] + 1;

        reuse_body.assign_size(size);
        reuse_body.compute_inertia();
    }
}


// Mass Decrease
if (obj == 7)              // PANSAT
{
    if ((mx > 113 && mx < 129) && (my > 866 && my < 883))
    {
        if (mass > 31.0)
            mass = mass - 1;

        reuse_body.assign_mass(mass);
        reuse_body.compute_inertia();
    }
```

```
}

else                                    // all other shapes
{
    if ((mx > 113 && mx < 129) && (my > 866 && my < 883))
    {
        if (mass > 50.0)
            mass = mass - 50;

        reuse_body.assign_mass(mass);
        reuse_body.compute_inertia();
    }
}

// Mass Increase
if (obj == 7)           // PANSAT
{
    if ((mx > 165 && mx < 181) && (my > 866 && my < 883))
    {
        mass = mass + 1;
        reuse_body.assign_mass(mass);
        reuse_body.compute_inertia();
    }
}

else
{
    if ((mx > 165 && mx < 181) && (my > 866 && my < 883))
    {
        mass = mass + 50;
        reuse_body.assign_mass(mass);
        reuse_body.compute_inertia();
    }
}

// Rotation Angle Decrease
if (mx > 379 && mx < 396)
{
    if (my > 936 && my < 953)       // angle about x
    {
        if (obj == 7)
            theta[0] = theta[0] - 0.1;
        else
```

```
            theta[0] = (int (theta[0]) - 5) % 360;
        }

        if (my > 923 && my < 937)       // angle about y
        {
            if (obj == 7)
                theta[1] = theta[1] - 0.1;
            else
                theta[1] = (int (theta[1]) - 5) % 360;
        }

        if (my > 909 && my < 924)       // angle about z
        {
            if (obj == 7)
                theta[2] = theta[2] - 0.1;
            else
                theta[2] = (int (theta[2]) - 5) % 360;
        }
    }


// Rotation Angle Increase
if (mx > 433 && mx < 448)
{
    if (my > 936 && my < 953)       // angle about x
    {
        if (obj == 7)
            theta[0] = theta[0] + 0.1;
        else
            theta[0] = (int (theta[0]) + 5) % 360;
    }

    if(my > 923 && my < 937)        // angle about y
    {
        if (obj == 7)
            theta[1] = theta[1] + 0.1;
        else
            theta[1] = (int (theta[1]) + 5) % 360;
    }

    if(my > 909 && my < 924)        // angle about z
    {
```

```
        if (obj == 7)
            theta[2] = theta[2] + 0.1;
        else
            theta[2] = (int (theta[2]) + 5) % 360;
    }
}

// Angular Velocity Magnitude Decrease
if ((mx > 186 && mx < 202) && (my > 866 && my < 883))
    vel_mag = vel_mag / 10;

// Angular Velocity Magnitude Increase
if ((mx > 253 && mx < 269) && (my > 866 && my < 883))
    vel_mag = vel_mag * 10;

// Moment Magnitude Decrease
if ((mx > 279 && mx < 296) && (my > 866 && my < 883))
    mom_mag = mom_mag / 10;

// Monment Magnitude Increase
if ((mx > 346 && mx < 361) && (my > 866 && my < 883))
    mom_mag = mom_mag * 10;

// Altitude Decrease
if ((mx > 373 && mx < 388) && (my > 866 && my < 883))
{
    if (obj == 7)
    {
        if (altitude > 0)
            altitude = altitude - 25;
    }
    else
    {
        if (altitude > 0)
            altitude = altitude - 500;
    }

    // compute inertial angular velocity (radians/sec)
    if (altitude != 0)
    {
        radius = earth_radius + altitude;
        omega = sqrt(mu_earth / (radius*radius*radius));
        omega_vec[2] = omega;z = omega_vec[2];
```

```
    }
    else
    {
        omega = 0;
        omega_vec[2] = 0;z = 0;
    }
}

// Altitude Increase
if ((mx > 438 && mx < 456) && (my > 866 && my < 883))
{
    if (obj == 7)
    {
        if (altitude < 99975)
            altitude = altitude + 25;
    }
    else
    {
        if (altitude < 99500)
            altitude = altitude + 500;
    }

    // compute inertial angular velocity (radians/sec)
    radius = earth_radius + altitude;
    omega = sqrt(mu_earth / (radius*radius*radius));
    omega_vec[2] = omega;
    z = omega_vec[2];
}

//Select Shape
if (mx > 19 && mx < 103)
{
    if (my > 936 && my < 953)        // select ball
    {
        obj = 1;
        reuse_body.assign_shape(ball.return_shape());
        reuse_body.assign_type(2);
        reuse_body.assign_size(size);
        reuse_body.assign_mass(ball.return_mass());
        reuse_body.compute_inertia();
        mass = reuse_body.return_mass();
    }
```

```
if (my > 923 && my < 937)        // select cube
{
    obj = 2;
    reuse_body.assign_shape(cube.return_shape());
    reuse_body.assign_type(1);
    reuse_body.assign_size(size);
    reuse_body.assign_mass(cube.return_mass());
    reuse_body.compute_inertia();
    mass = reuse_body.return_mass();
}

if (my > 909 && my < 924)        // select cylinder
{
    obj = 3;
    reuse_body.assign_shape(cylinder.return_shape());
    reuse_body.assign_type(3);
    reuse_body.assign_size(size);
    reuse_body.assign_mass(cylinder.return_mass());
    reuse_body.compute_inertia();
    mass = reuse_body.return_mass();
}

if (my > 897 && my < 910)        // add boom to existing body
{
    if (obj == 1)                // sphere and boom
    {
        obj = 4;
        reuse_body.assign_shape(ball_boom.return_shape());
        reuse_body.assign_type(71);size = boom_size;
        reuse_body.assign_size(size);
        reuse_body.assign_mass(ball_boom.return_mass());
        reuse_body.assign_inertia(ball_boom.return_inertia());
        inertia = reuse_body.return_inertia();
        mass = reuse_body.return_mass();
    }

    if (obj == 2)                // cube and boom
    {
        obj = 5;
        reuse_body.assign_shape(cube_boom.return_shape());
        reuse_body.assign_type(72);
        size = boom_size;
        reuse_body.assign_size(size);
```

```
            reuse_body.assign_mass(cube_boom.return_mass());
            reuse_body.assign_inertia(cube_boom.return_inertia());
            inertia = reuse_body.return_inertia();
            mass = reuse_body.return_mass();
        }

        if (obj == 3)                    // cylinder and boom
        {
            obj = 6;
            reuse_body.assign_shape(cylinder_boom.return_shape());
            reuse_body.assign_type(73)
            size = boom_size;
            reuse_body.assign_size(size);
            reuse_body.assign_mass(cylinder_boom.return_mass());
            reuse_body.assign_inertia(cylinder_boom.return_inertia());
            inertia = reuse_body.return_inertia();
            mass = reuse_body.return_mass();
        }
    }

    if (my > 884 && my < 898)        // select PANSAT
    {
        obj = 7;
        reuse_body.assign_shape(pansat.return_shape());
        reuse_body.assign_type(60);
        size = pansat_size;
        reuse_body.assign_size(size);
        reuse_body.assign_mass(pansat.return_mass());
        reuse_body.assign_inertia(pansat.return_inertia());
        inertia = reuse_body.return_inertia();
        mass = reuse_body.return_mass();
        theta.zero();
        altitude = 0;
    }
}

// Rotate Body button selected
if (mx > 640 && mx < 707 && my > 890 &&
    my < 959 &&  !NO_GO)
{
    // compute the DCM
    mat1.DCM_x_rotation(theta[0] * (pi / 180));
    mat2.DCM_y_rotation(theta[1] * (pi / 180));
```

```
    mat3.DCM_z_rotation(theta[2] * (pi / 180));
    rotation = rotation * mat3;
    rotation = rotation * mat2;
    rotation = rotation * mat1;

    // initial spin is a function of the inertial angular velocity
    // about the 3 axes
    initial_omega = rotation * omega_vec;
    x = initial_omega[0];
    y = initial_omega[1];
    z = initial_omega[2];

    GO = 21;
    step = 0.0;
}

// Inertial Moment button selected
if (mx > 740 && mx < 813 && my > 890 &&
    my < 959 &&  !NO_GO)
{
    GO = 11;
    elapsed_time = 0.0;
    step = 0.0;
}

// Spin Up button selected
if (mx > 840 && mx < 907 && my > 890 &&
    my < 959 && !NO_GO)
{
    GO = 1;
    NO_GO = 1;
}

if (GO == 2)
    NO_GO = 0;

// RESET button selected
if (mx > 940 && mx < 1007 && my > 890 && my < 959)
{
    reuse_body.zero();
    reuse_body.assign_size(size);
    mass = reuse_body.return_mass();
    inertia = reuse_body.return_inertia();
```

```
            x = 0;
            y = 0;
            z = 0;
            mox = 0;
            moy = 0;
            moz = 0;
            vel_mag = 1.0;
            mom_mag = 1.0;
            theta.zero();
            GO = 0;
            NO_GO = 0;
            step = 0.0;
            elapsed_time = 0.0;
            altitude = 0;
            omega = 0;
            omega_vec.zero();
            initial_omega.zero();
            mat1.reset();
            mat2.reset();
            mat3.reset();
            rotation.reset();
        }
    }
}

main_window();

// this section rotates the body from its inertial frame
if (GO == 21)       // reparation for 1st rotation
{
    reuse = reuse * 0;
    if(theta[0])
        reuse[axis1] = .3 * (theta[0] / fabs(theta[0]));

    // set ang velocity
    rot = theta[0] * (pi / 180) / 2;
    GO++;
    go_next = 0;
}

if(GO == 22)        // animates body
{
    if(go_next)     // stop body for second rotation
```

```
        {
            GO++;
            reuse_body.assign_ang_velocity_bc(0,0,0);
        }

        else
        {
            if((theta[0] > 0 && rot > .3 * read_delta()) || (theta[0] < 0 &&
                rot < -.3 * read_delta()))
            {                                       // rotation incomplete
                reuse_body.assign_ang_velocity_bc(reuse);
                rot = rot - reuse[axis1] * read_delta();
            }

            else        // finish 1st rotation
            {
                if(theta[0])
                    reuse = reuse * (rot / (reuse[axis1] * read_delta()));
                reuse_body.assign_ang_velocity_bc(reuse);
                go_next = 1;
            }
        }
    }

    if(GO == 23)        // prep for 2nd rotation
    {
        reuse = reuse * 0;
        if(theta[1])
            reuse[axis2] = .3 * (theta[1] / fabs(theta[1]));
        rot = theta[1] * (pi / 180) /2;
        GO++;
        go_next = 0;
    }

    if(GO == 24)        // animate 2nd rotation
    {
        if(go_next)     // stop for 3rd rotation
        {
            GO++;
            reuse_body.assign_ang_velocity_bc(0,0,0);
        }

        else
```

```
    {
        if((theta[1] > 0 && rct > .3 * read_delta()) || (theta[1] < 0 &&
          . rot < -.3 * read_delta()))
        {
            reuse_body.assign_ang_velocity_bc(reuse);
            rot = rot - reuse[axis2] * read_delta();
        }

        else        // finish 2nd rotation
        {
            if(theta[1])
                reuse = reuse * (rot / (reuse[axis2] * read_delta()));
            reuse_body.assign_ang_velocity_bc(reuse);
            go_next = 1;
        }
    }
}

if(GO == 25)        //prep for 3rd rotation
{
    reuse = reuse * 0;
    if(theta[2])
        reuse[axis3] = .3 * (theta[2] / fabs(theta[2]));
    rot = theta[2] * (pi / 180) / 2;
    GO++;
    go_next = 0;
}

if(GO == 26)        // animate 3rd rotation
{
    if(go_next)
    {
        reuse_body.assign_ang_velocity_bc(0,0,0);
        GO++;
    }

    else
    {
        if((theta[2] > 0 && rot > .3 * read_delta()) || (theta[2] < 0 &&
            rot < -.3 * read_delta()))
        {
            reuse_body.assign_ang_velocity_bc(reuse);
            rot = rot - reuse[axis3] * read_delta();
```

```
        }

        else        // finish 3rd rotation
        {
            if(theta[2])
                reuse = reuse * (rot / (reuse[axis3] * read_delta()));
            reuse_body.assign_ang_velocity_bc(reuse);
            go_next = 1;
        }
    }
}

if (GO > 11)        // for rotations only
{
    reuse_body.update_state_rk4();
    reuse_body.display();
}


if (GO == 1)   // spin body
{
    reuse_body.assign_ang_velocity_bc(x * vel_mag, y * vel_mag,
                                      z * vel_mag);
    GO++;
}

if (GO == 11)                          // set moment in inertial coordinates
{
    mox = 3 * omega * omega * ((reuse_body.return_inertia())[2]-
          (reuse_body.return_inertia())[1]) * rotation[1] * rotation[2];

    moy = 3 * omega * omega * ((reuse_body.return_inertia())[0]-
          (reuse_body.return_inertia())[2]) * rotation[0] * rotation[2];

    moz = 3 * omega * omega * ((reuse_body.return_inertia())[1]-
          (reuse_body.return_inertia())[0]) * rotation[1] * rotation[0];

    reuse_body.assign_moment(mox * mom_mag, moy * mom_mag,
                                      moz * mom_mag);

    elapsed_time = elapsed_time + step;
}
```

```
            if ((GO == 11) && duration < 0)              // apply moment
            {
                GO = 0;
                NO_GO = 0;
            }

            frame.display();
            if (duration > 0 && duration < step)         // last integration step
                set_delta(duration);

            step = reuse_body.update_state_rk45(.000001);
            reuse_body.display();
            angular_velocity = reuse_body.return_ang_velocity_bc();
            inertia = reuse_body.return_inertia();

            am1 = angular_velocity[0] * inertia[0];
            am2 = angular_velocity[1] * inertia[1];
            am3 = angular_velocity[2] * inertia[2];
            am = sqrt(am1 * am1 + am2 * am2 + am3 * am3);

            // display statistics
            stat_controls(angular_velocity[0],angular_velocity[1],angular_velocity[2],
                        am,reuse_body.return_mass(), (reuse_body.return_inertia())[0],
                        (reuse_body.return_inertia())[1],(reuse_body.return_inertia())[2]);
        }
}
```

APPENDIX B. GRAPHICS CODE

A. HEADER FILE

```
#ifndef GRAPHICS_H
#define GRAPHICS_H
#include <math.h>
#include "vector3D.H"
#include "matrix3x3.H"
#include "quaternion.H"
#include "rdobj_opcodes.h"
#include "rdobj_funcs.h"
#include <stdio.h>
#include <gl.h>
#include <device.h>

//initializes the graphic system
void initialize();

//initializes control window
void init_control_window();

//make viewing window active
void main_window();

// makes control window active
void control_window();

//clears control window
void clear_control_window();

//control window for euler program
void euler_controls(int, int, int, int, int, int,int,quaternion, double);

//control window for gyro program
void gyro_controls(double, double, double, int,double, vector3D, vector3D, double,
                   double, double, double,double, double, double, vector3D);

//statisic display for gyro program
void stat_controls(double, double, double, double,double, double, double, double);

//control window for frame program
void frame_controls(int, int, vector3D, vector3D,vector3D, int);
```

```
//standard function for viewing a scene
void view();

//used to view the scene for a point of view fixed to a rigid body
void view(quaternion, vector3D, int);

//attaches the eye to a rigid body
void attach_eye_to(vector3D*, int*);

//attaches the tatget to a rigid body
void attach_target_to(vector3D*);

//attaches the eye to a rigid body
void set_eye_to(double, double, double);

//attaches the target to a rigid body
void set_target_to(double, double, double);

//rotates the view in tenths of degrees
void rotate_view(int);

//displays the body axes of a rigid_body
void view_axis();

//gravity check - returns non zero value when gravity is turned on
int gravity_status();
void set_gravity_on();
void set_gravity_off();
void toggle_gravity();

//air resistance check - returns non zero value when air resistance is turned on
int air_resistance_status();
void set_air_resistance_on();
void set_air_resistance_off();
void toggle_air_resistance();

// c routines the must be accessed
extern "C"
{
        extern OBJECT* read_object(char[]);
        extern void ready_object_for_display( OBJECT* );
        extern void display_this_object( OBJECT* );
```

```
        };

#endif
```

B. SOURCE FILE

```
#ifndef GRAPHICS_C
#define GRAPHICS_C
#include "graphics.H"

#define NEARDEPTH 0x000000      /* the near and far planes used for Zbuffering*/
#define FARDEPTH  0x7fffff

OBJECT *lightobj, *axis;

//eye and target are the global variables that control the view point
//and reference point of the scene respectively
vector3D *eye = new vector3D(10.0, 10.0, 10.0), *target = new vector3D;
int *eye_display_field = NULL;
int gravity_flag = 0, air_resistance_flag = 0;
int twist = 0;
long main_win, control_win;

Matrix un = { 1.0, 0.0, 0.0, 0.0,
              0.0, 1.0, 0.0, 0.0,
              0.0, 0.0, 1.0, 0.0,
              0.0, 0.0, 0.0, 1.0};


int gravity_status()
{
        return gravity_flag;
}


void set_gravity_on()
{
        gravity_flag = 1;
}


void set_gravity_off()
{
```

```
        gravity_flag = 0;
}


void toggle_gravity()
{
        if(gravity_flag)
        {
                gravity_flag = 0;
        }
        else
        {
                gravity_flag = 1;
        }
}


int air_resistance_status()
{
        return air_resistance_flag;
}


void set_air_resistance_on()
{
        air_resistance_flag = 1;
}


void set_air_resistance_off()
{
        air_resistance_flag = 0;
}


void toggle_air_resistance()
{
        if (air_resistance_flag)
        {
                air_resistance_flag = 0;
        }
        else
        {
```

```
                air_resistance_flag = 1;
        }
}


void initialize()
{
        /* set up the preferred aspect ratio */
        keepaspect(XMAXSCREEN+1,YMAXSCREEN+1);
        prefsize(XMAXSCREEN/2,YMAXSCREEN/2);
        prefposition(0,XMAXSCREEN * 0.8 ,0,YMAXSCREEN * 0.8);

        /* open a window for the program */
        main_win = winopen("Main");
        wintitle("Gravity-Gradient Visualizer, By Jeff Stewart");

        /* put the IRIS into double buffer mode */
        doublebuffer();

        /* put the iris into rgb mode */
        RGBmode();

        /* configure the IRIS (means use the above command settings) */
        gconfig();

        /* set the depth for z-buffering */
        lsetdepth(NEARDEPTH,FARDEPTH);

        /* queue the redraw device */
        qdevice(REDRAW);

        /*queue the menu button*/
        qdevice(MENUBUTTON);

        /*turn the cursor on*/
        curson();

        /*select gouraud shading*/
        shademodel(GOURAUD);

        /*turn on the new projection matrix mode*/
        mmode(MVIEWING);
```

```
        /*Turn on Zbuffering*/
        zbuffer(TRUE);
        lightobj = read_object("the_light.off");
        axis = read_object("axis.off");
        ready_object_for_display(lightobj);
        ready_object_for_display(axis);

        //queue up input devices
        qdevice(LEFTMOUSE);
        qdevice(RIGHTMOUSE);
        qdevice(MOUSEX);
        qdevice(MOUSEY);
        qdevice(RIGHTARROWKEY);
        qdevice(LEFTARROWKEY);
        qdevice(UPARROWKEY);
        qdevice(DOWNARROWKEY);
        qdevice(SPACEKEY);
        qdevice(EQUALKEY);
        qdevice(MINUSKEY);
        qdevice(F1KEY);
        qdevice(F2KEY);
        qdevice(F3KEY);
        qdevice(F4KEY);
        qdevice(F5KEY);
        qdevice(F6KEY);
        qdevice(F7KEY);
        qdevice(F8KEY);
        qdevice(F9KEY);
        qdevice(F10KEY);
        qdevice(F11KEY);
        qdevice(F12KEY);

        //clear draw and display buffer
        czclear(0xFFFF7200,getgdesc(GD_ZMAX));
        swapbuffers();
        czclear(0xFFFF7200,getgdesc(GD_ZMAX));
}


void init_control_window()
{
        /* set up the preferred aspect ratio */
        prefposition(0,XMAXSCREEN * 0.8,YMAXSCREEN * 0.87, YMAXSCREEN);
```

```
        /* open a window for the program */
        control_win = winopen("control");
        wintitle("System Control Window");

        /* put the IRIS into double buffer mode */
        doublebuffer();
        RGBmode();
        gconfig();
        pushmatrix();
        ortho2(0.0, 769.0, 0.0, 100.0);
        RGBcolor(255,255,255);
        clear();
        popmatrix();
        swapbuffers();
}


void main_window()
{
        winset(main_win);
}


void control_window()
{
        winset(control_win);
}


void clear_control_window()
{
        winset(control_win);
        pushmatrix();
        ortho2(0.0, 769.0, 0.0, 100.0);
        RGBcolor(255,255,255);
        clear();
        popmatrix();
        swapbuffers();
        winset(main_win);
}
```

```
void gyro_controls(double x, double y, double z, int object, double altitude, vector3D
size,                          vector3D theta, double mass, double t1, double t2, double
t3,                            double vel_mag, double mom_mag, double elapsed,
                       vector3D inertia)
{
        float pt [3][2] = { {142,48},
                            {148,48},
                            {145,42}};          // down arrow starting coordinates, begins
on                                              // Z velocity
        float pt2 [3][2] = { {182,42},
                             {188,42},
                             {185,48}};         // up arrow starting coodinates, begins on Z
                                                // velocity

        char s[32];

        winset(control_win);
        pushmatrix();
        ortho2(0.0, 769.0, 0.0, 100.0);
        RGBcolor(255,255,255);
        clear();

        //Go & Reset Buttons
        RGBcolor(200,200,200);
        rectf(475.0, 25.0, 525.0, 75.0);
        rectf(550.0, 25.0, 605.0, 75.0);
        rectf(625.0, 25.0, 675.0, 75.0);
        rectf(700.0, 25.0, 750.0, 75.0);
        RGBcolor(0,0,0);
        cmov2(482, 50);
        charstr("Rotate");
        cmov2(487, 40);
        charstr("Body");
        cmov2(555, 50);
        charstr("Gravity");
        cmov2(552, 40);
        charstr("Gradient");
        cmov2(639,50);
        charstr("Spin");
        cmov2(643,40);
        charstr("up");
        cmov2(710,45);
        charstr("Reset");
```

```
// Angular Velocity
RGBcolor(0,0,255);
rectf(140.0, 40.0, 190.0, 70.0);
RGBcolor(255,255,0);
cmov2(142, 61);
sprintf(s, "%.4f", (double) x);
charstr(s);
cmov2(142, 51);
sprintf(s, "%.4f", (double) y);
charstr(s);
cmov2(142, 41);
sprintf(s, "%.4f", (double) z);
charstr(s);

// External Moment
RGBcolor(0,0,255);
rectf(210.0, 40.0, 260.0, 70.0);
RGBcolor(255,255,0);
cmov2(212, 61);
sprintf(s, "%.4f", t1);
charstr(s);
cmov2(212, 51);
sprintf(s, "%.4f", t2);
charstr(s);
cmov2(212, 41);
sprintf(s, "%.4f", t3);
charstr(s);

// Rotation angle, Theta
RGBcolor(0,0,255);
rectf(290.0, 40.0, 320.0, 70.0);
RGBcolor(200,200,200);
rectf(280.0, 40.0, 290.0, 70.0);
rectf(320.0, 40.0, 330.0, 70.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 140.0;
pt[1][0] = pt[1][0] + 140.0;
pt[2][0] = pt[2][0] + 140.0;
pt2[0][0] = pt2[0][0] + 140.0;
pt2[1][0] = pt2[1][0] + 140.0;
pt2[2][0] = pt2[2][0] + 140.0;
```

```
polf2(3,pt);polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);

if (object == 7)                    // PANSAT
{
        cmov2(292, 61);
        sprintf(s, "%.1f", theta[0]);
        charstr(s);
        cmov2(292, 51);
        sprintf(s, "%.1f", theta[1]);
        charstr(s);
        cmov2(292, 41);
        sprintf(s, "%.1f", theta[2]);
        charstr(s);
}
else                                // all other shapes
{
        cmov2(292, 61);
        sprintf(s, "%.0f", theta[0]);
        charstr(s);
        cmov2(292, 51);
        sprintf(s, "%.0f", theta[1]);
        charstr(s);
        cmov2(292, 41);
        sprintf(s, "%.0f", theta[2]);
        charstr(s);
}
```

```
//Clock
RGBcolor(255,0,0);
rectf(355.0, 55.0, 460.0, 85.0);
RGBcolor(0,0,0);
cmov2(365.0,71.0);
charstr("Elapsed Time");
RGBcolor(255,255,255);
cmov2(390, 60);
sprintf(s, "%.1f", elapsed);
charstr(s);
RGBcolor(0,0,0);
cmov2(10.0,90.0);
charstr("Gravity Gradient Control Window");
cmov2(10.0,75.0);
charstr("Shape");

 if (object > 3)                    // PANSAT or boom object
{
        cmov2(82.0,75.0);
        charstr("Inertia");
}
else                                // cube, sphere, or cylinder
{
        cmov2(92.0,75.0);
        charstr("Size");
}
cmov2(143.0,75.0);
charstr("Ang Vel");
RGBcolor(255,0,0);
cmov2(197.0,61.5);
charstr("X");
RGBcolor(0,0,255);
cmov2(197.0,51.5);
charstr("Y");
RGBcolor(0,0,0);
cmov2(197.0,41.5);
charstr("Z");
cmov2(215.0,75.0);
charstr("Moment");
cmov2(288.0,75.0);
charstr("Theta");
cmov2(92.0,23.0);
charstr("Mass");
```

```
cmov2(143.0,23.0);
charstr("Vel Mag");
cmov2(211.0,23.0);
charstr("Mom Mag");
cmov2(280.0,23.0);
charstr("Altitude");

// Select between the different base objects
RGBcolor(0,0,255);
rectf(10.0, 20.0, 70.0, 70.0);
RGBcolor(255,255,0);
cmov2(15.0,61.5);
charstr("Sphere");
cmov2(15.0,51.5);
charstr("Block");
cmov2(15.0,41.5);
charstr("Cylinder");
cmov2(15.0,31.5);
charstr("Add Boom");
cmov2(15.0,21.5);
charstr("PANSAT");

switch(object)
{
        case 1:
                RGBcolor(255,255,0);
                rectf(10.0, 60.0, 70.0, 70.0);
                RGBcolor(0,0,255);
                cmov2(15.0,61.5);
                charstr("Sphere");
                break;

        case 2:
                RGBcolor(255,255,0);
                rectf(10.0, 50.0, 70.0, 60.0);
                RGBcolor(0,0,255);
                cmov2(15.0,51.5);
                charstr("Block");
                break;

        case 3:
                RGBcolor(255,255,0);
                rectf(10.0, 40.0, 70.0, 50.0);
```

```
            RGBcolor(0,0,255);
            cmov2(15.0,41.5);
            charstr("Cylinder");
            break;

    case 4:
            RGBcolor(255,255,0);
            rectf(10.0, 60.0, 70.0, 70.0);
            rectf(10.0, 30.0, 70.0, 40.0);
            RGBcolor(0,0,255);
            cmov2(15.0,61.5);
            charstr("Sphere");
            cmov2(15.0,31.5);
            charstr("Add Boom");
            break;

    case 5:
            RGBcolor(255,255,0);
            rectf(10.0, 50.0, 70.0, 60.0);
            rectf(10.0, 30.0, 70.0, 40.0);
            RGBcolor(0,0 255);
            cmov2(15.0,51.5);
            charstr("Block");
            cmov2(15.0,31.5);
            charstr("Add Boom");
            break;

    case 6:
            RGBcolor(255,255,0);
            rectf(10.0, 40.0, 70.0, 50.0);
            rectf(10.0, 30.0, 70.0, 40.0);
            RGBcolor(0,0,255);
            cmov2(15.0,41.5);
            charstr("Cylinder");
            cmov2(15.0,31.5);
            charstr("Add Boom");
            break;

    case 7:
            RGBcolor(255,255,0);
            rectf(10.0, 20.0, 70.0, 30.0);
            RGBcolor(0,0,255);
            cmov2(15.0,21.5);
```

```
                charstr("PANSAT");
                break;

        default:
                RGBcolor(255,255,0);
                rectf(10.0, 50.0, 70.0, 60.0);
                RGBcolor(0,0,255);
                cmov2(15.0,51.5);
                charstr("Block");
                break;
}

// Object Size
RGBcolor(0,0,255);
rectf(90.0, 40.0, 120.0, 70.0);
RGBcolor(200,200,200);
rectf(80.0, 40.0, 90.0, 70.0);
rectf(120.0, 40.0, 130.0, 70.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] - 200.0;
pt[1][0] = pt[1][0] - 200.0;
pt[2][0] = pt[2][0] - 200.0;
pt2[0][0] = pt2[0][0] - 200.0;
pt2[1][0] = pt2[1][0] - 200.0;
pt2[2][0] = pt2[2][0] - 200.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt); polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt); polf2(3,pt2);
RGBcolor(255,255,0);
```

```
if (object == 7)                                    // Inertia for PANSAT
{
        RGBcolor(255,255,0);
        cmov2(92, 61);
        sprintf(s, "%.2f", inertia[0]);
        charstr(s);
        cmov2(92, 51);
        sprintf(s, "%.2f", inertia[1]);
        charstr(s);
        cmov2(92, 41);
        sprintf(s, "%.2f", inertia[2]);
        charstr(s);
}
else if ((object > 3) && (object < 7)) // Inertia for boom object
{
        RGBcolor(255,255,0);
        cmov2(92, 61);
        sprintf(s, "%.0f", inertia[0]);
        charstr(s);
        cmov2(92, 51);
        sprintf(s, "%.0f", inertia[1]);
        charstr(s);
        cmov2(92, 41);
        sprintf(s, "%.0f", inertia[2]);
        charstr(s);
}
else                                        // size for cube, sphere, or cylinder
{
        cmov2(92, 61);
        sprintf(s, "%.0f", size[0]);
        charstr(s);
        cmov2(92, 51);
        sprintf(s, "%.0f", size[1]);
        charstr(s);
        cmov2(92, 41);
        sprintf(s, "%.0f", size[2]);
        charstr(s);
}

//Object Mass
RGBcolor(0,0,255);
rectf(90.0, 8.0, 120.0, 18.0);
```

```
RGBcolor(200,200,200);
rectf(80.0, 8.0, 90.0, 18.0);
rectf(120.0, 8.0, 130.0, 18.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][1] = pt[0][1] - 32.0;
pt[1][1] = pt[1][1] - 32.0;
pt[2][1] = pt[2][1] - 32.0;
pt2[0][1] = pt2[0][1] - 32.0;
pt2[1][1] = pt2[1][1] - 32.0;
pt2[2][1] = pt2[2][1] - 32.0;
polf2(3,pt); polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(92, 10);
sprintf(s, "%.0f", mass);
charstr(s);

// Angular Velocity Magnitude
RGBcolor(0,0,255);
rectf(145.0, 8.0, 185.0, 18.0);
RGBcolor(200,200,200);
rectf(135.0, 8.0, 145.0, 18.0);
rectf(185.0, 8.0, 195.0, 18.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 55.0;
pt[1][0] = pt[1][0] + 55.0;
pt[2][0] = pt[2][0] + 55.0;
pt2[0][0] = pt2[0][0] + 65.0;
pt2[1][0] = pt2[1][0] + 65.0;
pt2[2][0] = pt2[2][0] + 65.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(147, 9);
sprintf(s, "%.0e", vel_mag);
charstr(s);

// External Moment Magnitude
RGBcolor(0,0,255);
rectf(215.0, 8.0, 255.0, 18.0);
RGBcolor(200,200,200);
rectf(205.0, 8.0, 215.0, 18.0);
```

```
        rectf(255.0, 8.0, 265.0, 18.0);
        // Draw up and down arrows
        RGBcolor(0,0,0);
        pt[0][0] = pt[0][0] + 70.0;
        pt[1][0] = pt[1][0] + 70.0;
        pt[2][0] = pt[2][0] + 70.0;
        pt2[0][0] = pt2[0][0] + 70.0;
        pt2[1][0] = pt2[1][0] + 70.0;
        pt2[2][0] = pt2[2][0] + 70.0;
        polf2(3,pt);
        polf2(3,pt2);
        RGBcolor(255,255,0);
        cmov2(217, 9);
        sprintf(s, "%.0e", mom_mag);
        charstr(s);

        // Object altitude
        RGBcolor(0,0,255);
        rectf(285.0, 8.0, 325.0, 18.0);
        RGBcolor(200,200,200);
        rectf(275.0, 8.0, 285.0, 18.0);
        rectf(325.0, 8.0, 335.0, 18.0);
        // Draw up and down arrows
        RGBcolor(0,0,0);
        pt[0][0] = pt[0][0] + 70.0;
        pt[1][0] = pt[1][0] + 70.0;
        pt[2][0] = pt[2][0] + 70.0;
        pt2[0][0] = pt2[0][0] + 70.0;
        pt2[1][0] = pt2[1][0] + 70.0;
        pt2[2][0] = pt2[2][0] + 70.0;
        polf2(3,pt);
        polf2(3,pt2);
        RGBcolor(255,255,0);
        cmov2(287, 9);
        sprintf(s, "%.0f", altitude);
        charstr(s);
        popmatrix();
        swapbuffers();
}


void stat_controls(double x, double y, double z, double am, double mass, double Ix,
                   double Iy,  double Iz)
```

```
{
    winset(main_win);
    char s[32];
    RGBcolor(0,0,0);
    RGBcolor(255,0,0);
    cmovs(25, 35, 0);
    charstr("X velocity");
    cmovs(37, 35, 0);
    sprintf(s, "%.5f", x);
    charstr(s);
    cmovs(-50, 35, 0);
    charstr("Ixx");
    cmovs(-45, 35, 0);
    sprintf(s, "%.5f", Ix);
    charstr(s);
    RGBcolor(0,0,255);
    cmovs(25, 33, 0);
    charstr("Y velocity");
    cmovs(37, 33, 0);
    sprintf(s, "%.5f", y);
    charstr(s);
    cmovs(-50, 33, 0);
    charstr("Iyy");
    cmovs(-45, 33, 0);
    sprintf(s, "%.5f", Iy);
    charstr(s);
    RGBcolor(0,0,0);
    cmovs(25, 31, 0);
    charstr("Z velocity");
    cmovs(37, 31, 0);
    sprintf(s, "%.5f", z);
    charstr(s);
    cmovs(-50, 31, 0);
    charstr("Izz");
    cmovs(-45, 31, 0);
    sprintf(s, "%.5f", Iz);
    charstr(s);
    cmovs(25, 27, 0);
    charstr("Ang Momentum");
    cmovs(39, 27, 0);
    sprintf(s, "%.3f", am);
    charstr(s);
    cmovs(-50, 27, 0);
```

```
        charstr("Mass");
        cmovs(-45, 27, 0);
        sprintf(s, "% 3f", mass);
        charstr(s);
}


void attach_eye_to(vector3D* v, int* i)
{
        if (eye_display_field != NULL)
        {
                *eye_display_field = 1;
        }
        eye = v;
        eye_display_field = i;
        *eye_display_field = 0;
        twist = 0;
}


void attach_target_to(vector3D* v)
{
        target = v;
        twist = 0;
}


void set_eye_to(double x, double y, double z)
{
        if (eye_display_field != NULL)
        {
                *eye_display_field = 1;
                eye_display_field = NULL;
        }
        eye = new vector3D(x, y, z);
        twist = 0;
}


void set_target_to(double x, double y, double z)
{
        target = new vector3D(x, y, z);
        twist = 0;
```

```
}


void view()
{
        swapbuffers();
        czclear(0xFFFF7200,getgdesc(GD_ZMAX));
        loadmatrix(un);
        perspective(450,1.25,0.2,10000.0);
        lookat((*eye)[0],(*eye)[1],(*eye)[2],(*target)[0],(*target)[1],(*target)[2],
                (int) (twist * 572.957795131));
        display_this_object(lightobj);
}


void view(quaternion q, vector3D new_eye, int view_axis)
{
        Matrix rt = {   1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, 1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0};

        matrix3x3 rotation, axis;
        swapbuffers();
        czclear(0xFFFF7200,getgdesc(GD_ZMAX));
        loadmatrix(un);
        perspective(450,1.25,0.2,10000.0);
        switch(view_axis)
        {
                //Negative Y axis
                case -2:
                        axis.DCM_x_rotation(1.5707963268);
                        break;

                //Negative X axis
                case -1:
                        axis.DCM_y_rotation(-1.5707963268);
                        break;

                //Positive X axis
                case 1:
                        axis.DCM_y_rotation(1.5707963268);
                        break;
```

```
            //Positive Y axis
            case 2:
                axis.DCM_x_rotation(-1.5707963268);
                break;

            //Positive Z axis
            case 3:
                axis.DCM_y_rotation(3.14159265359);
                break;

            default:
                break;
        }

        rotation.DCM_body_to_world(q);
        rotation = rotation * axis;
        new_eye = new_eye * -1;

        rt[0][0] = rotation[0]; rt[1][0] = rotation[3]; rt[2][0] = rotation[6]; rt[3][0] = 0.0;
        rt[0][1] = rotation[1]; rt[1][1] = rotation[4]; rt[2][1] = rotation[7]; rt[3][1] = 0.0;
        rt[0][2] = rotation[2]; rt[1][2] = rotation[5]; rt[2][2] = rotation[8]; rt[3][2] = 0.0;
        rt[3][0] = 0;   rt[3][1] = 0;   rt[3][2] = 0;   rt[3][3] = 1.0;
        multmatrix(rt);

        rt[0][0] = 1;   rt[1][0] = 0;   rt[2][0] = 0;   rt[3][0] = 0;
        rt[0][1] = 0;   rt[1][1] = 1;   rt[2][1] = 0;   rt[3][1] = 0;
        rt[0][2] = 0;   rt[1][2] = 0;   rt[2][2] = 1;   rt[3][2] = 0;
        rt[3][0] = new_eye[0];          rt[3][1] = new_eye[1];          rt[3][2] = new_eye[2];
        rt[3][3] = 1.0;                 multmatrix(rt);

        display_this_object(lightobj);
}


void view_axis()
{
        display_this_object(axis);
}


void rotate_view(int angle)
{
```

```
        twist = angle;
}

#endif
```

APPENDIX C. RIGID_BODY CODE

A. HEADER FILE

```
#ifndef RIGID_BODY_H
#define RIGID_BODY_H
#include <math.h>
#include "vector3D.H"
#include "quaternion.H"
#include "graphics.H"
#include "time.H"
#include "matrix3x3.H"
#include "constants.H"

class rigid_body
{
        double mass;
        vector3D *location;
        vector3D velocity;
        vector3D acceleration;
        vector3D force;
        quaternion orientation;
        vector3D ang_velocity;          // body coordinates
        vector3D ang_acceleration;      // body coordinates
        vector3D moment;                // body coordinates
        matrix3x3 inertia;
        vector3D size;
        double surface_area;
        OBJECT *shape;
        int display_axis;
        int *display_shape;
        int type_body;
        vector3D holder1;
        vector3D holder2;
        quaternion holder3;

public:
        void compute_inertia();
        rigid_body();
        rigid_body(int);
        rigid_body(char*);
        void assign_mass(double);
        void assign_size(double, double, double);
```

```
void assign_size(double);
void assign_size(vector3D);
void assign_surface_area(double);
void assign_inertia(double, double, double);
void assign_inertia(vector3D);
void assign_orientation(double, double, double, double);
void assign_orientation(quaternion);
void assign_shape(OBJECT*);
void assign_type(int);
void assign_holder1(double, double, double);
void assign_holder2(double, double, double);
void assign_holder3(double, double, double, double);
void assign_holder1(vector3D);
void assign_holder2(vector3D);
void assign_holder3(quaternion);

// Assign values to the items using doubles in world coordinates
void assign_location(double, double, double);
void assign_velocity(double, double, double);
void assign_acceleration(double, double, double);
void assign_ang_velocity(double, double, double);
void assign_ang_acceleration(double, double, double);
void assign_force(double, double, double);
void assign_moment(double, double, double);

// Assign values to the items using vector3D in world coordinates
void assign_location(vector3D);
void assign_velocity(vector3D);
void assign_acceleration(vector3D);
void assign_ang_velocity(vector3D);
void assign_ang_acceleration(vector3D);
void assign_force(vector3D);
void assign_moment(vector3D);

// Assign values to the items using doubles in body coordinates
void assign_velocity_bc(double, double, double);
void assign_acceleration_bc(double, double, double);
void assign_ang_velocity_bc(double, double, double);
void assign_ang_acceleration_bc(double, double, double);
void assign_force_bc(double, double, double);
void assign_moment_bc(double, double, double);

// Assign values to the items using vector3D in body coordinates
```

```
void assign_velocity_bc(vector3D);
void assign_acceleration_bc(vector3D);
void assign_ang_velocity_bc(vector3D);
void assign_ang_acceleration_bc(vector3D);
void assign_force_bc(vector3D);
void assign_moment_bc(vector3D);

// Return values of the items world coordinates
double return_mass();
vector3D return_inertia();
vector3D return_size();
vector3D return_location();
vector3D* return_location_ptr();
vector3D return_velocity();
vector3D return_acceleration();
quaternion return_orientation();
vector3D return_ang_velocity();
vector3D return_ang_acceleration();
vector3D return_force();
vector3D return_moment();
double return_surface_area();
OBJECT* return_shape();
int return_type();
vector3D return_holder1();
vector3D return_holder2();
quaternion return_holder3();

// Return values of the items body coordinates
vector3D return_velocity_bc();
vector3D return_acceleration_bc();
vector3D return_ang_velocity_bc();
vector3D return_ang_acceleration_bc();
vector3D return_force_bc();
vector3D return_moment_bc();
void display();
void update_state();
void update_state_rk4();
double update_state_rk45(double);
void add_axis();
void remove_axis();
void attach_eye();
void attach_target();
void attached_body_update(rigid_body);
```

```
        void zero();
        ~rigid_body()
        {
                delete shape;
        delete location;
        }
};

void set_eye(double, double, double);
void set_target(double, double, double);

#endif
```

B. SOURCE FILE

```
#ifndef RIGID_BODY_C
#define RIGID_BODY_C
#include "rigid_body.H"

void rigid_body::compute_inertia()
{
        switch(type_body)
        {
                case 1:
                        inertia[0] = mass * ((size[1] * size[1]) + (size[2] * size[2])) / 12.0;
                        inertia[4] = mass * ((size[0] * size[0]) + (size[2] * size[2])) / 12.0;
                        inertia[8] = mass * ((size[0] * size[0]) + (size[1] * size[1])) / 12.0;
                        break;

                case 2:
                        inertia[0] = mass * ((size[1] * size[1]) + (size[2] * size[2])) / 5.0;
                        inertia[4] = mass * ((size[0] * size[0]) + (size[2] * size[2])) / 5.0;
                        inertia[8] = mass * ((size[0] * size[0]) + (size[1] * size[1])) / 5.0;
                        break;

                case 3:
                        inertia[0] = mass * ((size[1] * size[1]) + (size[2] * size[2])) / 4.0;
                        inertia[4] = (mass * size[2] * size[2] / 4.0) +
                                     (mass * size[0] * size[0] /12.0);
                        inertia[8] = (mass * size[1] * size[1] / 4.0) +
                                     (mass * size[0] * size[0] / 12.0);
                        break;
        }
```

```
}


void rigid_body::assign_type(int t)
{
        type_body = t;
}


rigid_body::rigid_body()
{
        location = new vector3D;
        mass = 1000.0;
        size[0] = 1.0;
        size[1] = 1.0;
        size[2] = 1.0;
        surface_area = 1.0;
        shape = NULL;
        type_body = 0;
        display_axis = 0;
        display_shape = new int;
        *display_shape = 1;
}


rigid_body::rigid_body(char* off_file)
{
        location = new vector3D;
        mass = 1000.0;
        size[0] = 1.0;
        size[1] = 1.0;
        size[2] = 1.0;
        surface_area = 1.0;
        shape = read_object(off_file);
        ready_object_for_display(shape);
        type_body = 0;
        display_axis = 0;
        display_shape = new int;
        *display_shape = 1;
}


rigid_body::rigid_body(int n)
```

```
{
        location = new vector3D;
        mass = 1000.0;
        size[0] = 1.0;
        size[1] = 1.0;
        size[2] = 1.0;
        surface_area = 1.0;
        switch(n)
        {
                case 100:
                        shape = read_object("frame.off");
                        type_body = 100;
                        break;

                case 200:
                        shape = read_object("axis.off");
                        type_body = 200;
                        break;

                case 1:
                        shape = read_object("cube.off");
                        mass = 1000.0;
                        inertia[0] = 166.7;
                        inertia[4] = 166.7;
                        inertia[8] = 166.7;
                        type_body = 1;
                        surface_area = 3.0;
                        break;

                case 2:
                        shape = read_object("sphere.off");
                        type_body = 2;
                        mass = 1000.0;
                        inertia[0] = 100.0;
                        inertia[4] = 100.0;
                        inertia[8] = 100.0;
                        surface_area = .5;
                        break;

                case 3:
                        shape = read_object("cylender.off");
                        type_body = 3;
                        mass = 1000.0;
```

```
        inertia[0] = 500.0;
        inertia[4] = 333.3;
        inertia[8] = 333.3;
        surface_area = 2.0;
        break;

case 15:
        shape = read_object("f15.off");
        mass = 19076;
        surface_area = 5.46;
        type_body = 15;
        break;

case 23:
        shape = read_object("rubber_ban.off");
        mass = 100;
        surface_area = .01;
        type_body = 23;
        break;

case 31:
        shape = read_object("l1.off");
        mass = 100;
        surface_area = .01;
        type_body = 1;
        break;

case 32:
        shape =         read_object("l2.off");
        mass = 100;
        surface_area = .01;
        type_body = 1;
        break;

case 33:
        shape = read_object("l3.off");
        mass = 100;
        surface_area = .01;
        type_body = 1;
        break;

case 34:
        shape = read_object("l4.off");
```

```
        mass = 100;
        surface_area = .01;
        type_body = 1;
        break;

case 35:
        shape = read_object("l5.off");
        mass = 100;
        surface_area = .01;
        type_body = 1;
        break;

case 50:
        shape = read_object("shuttle.off");
        type_body = 50;
        mass = 1570.8;
        inertia[0] = 327.2;
        inertia[4] = 392.7;
        inertia[8] = 327.2;
        break;

case 60:
        shape = read_object("pansat.off");
        type_body = 60;
        mass = 46.14179;
        inertia[0] = 0.8914372;
        inertia[4] = 4.081316;
        inertia[8] = 4.082231;
        break;

case 71:
        shape = read_object("sphere_boom.off");
        mass = 1000.0;
        inertia[0] = 401.0;
        inertia[4] = 473.0;
        inertia[8] = 473.0;
        type_body = 71;
        break;

case 72:
        shape = read_object("cube_boom.off");
        mass = 1000.0;
        inertia[0] = 166.8;
```

```
                inertia[4] = 238.8;
                inertia[8] = 238.8;
                type_body = 72;
                break;

        case 73:
                shape = read_object("cylender_boom.off");
                mass = 1000.0;
                inertia[0] = 501.0;
                inertia[4] = 406.0;
                inertia[8] = 406.0;
                type_body = 73;
                break;

        case 90:
                shape = read_object("ground.off");
                type_body = 90;
                break;

        case 91:
                shape = read_object("floor.off");
                type_body = 91;
                break;

        default:
                shape = NULL;
                type_body = 0;
                break;
    }
    if (type_body)
        ready_object_for_display(shape);
    display_axis = 0;
    display_shape = new int;
    *display_shape = 1;
}


void rigid_body::assign_shape(OBJECT* o)
{
    shape = o;
}
```

```
void rigid_body::assign_mass(double n)
{
        mass = n;
}


void rigid_body::assign_surface_area(double n)
{
        surface_area = n;
}


void rigid_body::assign_size(double x, double y, double z)
{
        size[0] = x;
        size[1] = y;
        size[2] = z;
}


void rigid_body::assign_size(double x)
{
        size[0] = x;
        size[1] = x;
        size[2] = x;
}


void rigid_body::assign_size(vector3D v)
{
        size[0] = v[0];
        size[1] = v[1];
        size[2] = v[2];
}


 void rigid_body::assign_location(double x, double y, double z)
{
        (*location)[0] = x;
        (*location)[1] = y;
        (*location)[2] = z;
}
```

```
void rigid_body::assign_velocity(double x, double y, double z)
{
        velocity[0] = x;
        velocity[1] = y;
        velocity[2] = z;
}


void rigid_body::assign_acceleration(double x, double y, double z)
{
        acceleration[0] = x;
        acceleration[1] = y;
        acceleration[2] = z;
}


void rigid_body::assign_force(double x, double y, double z)
{
        force[0] = x;
        force[1] = y;
        force[2] = z;
}


void rigid_body::assign_orientation(double x, double y, double z, double w)
{
        orientation[0] = x;
        orientation[1] = y;
        orientation[2] = z;
        orientation[3] = w;
}


void rigid_body::assign_orientation(quaternion q)
{
        orientation[0] = q[0];
        orientation[1] = q[1];
        orientation[2] = q[2];
        orientation[3] = q[3];
}
```

```
void rigid_body::assign_inertia(double x, double y, double z)
{
        inertia[0] =x;
        inertia[4] =y;
        inertia[8] =z;
}


void rigid_body::assign_inertia(vector3D v)
{
        inertia[0] = v[0];
        inertia[4] = v[1];
        inertia[8] = v[2];
}


void rigid_body::assign_ang_velocity(double x, double y, double z)
{
        vector3D v(x, y, z);
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        ang_velocity[0] = v[0];
        ang_velocity[1] = v[1];
        ang_velocity[2] = v[2];
}


void rigid_body::assign_ang_acceleration(double x, double y, double z)
{
        vector3D v(x, y, z);
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        ang_acceleration[0] = v[0];
        ang_acceleration[1] = v[1];
        ang_acceleration[2] = v[2];
}


void rigid_body::assign_moment(double x, double y, double z)
{
        vector3D v(x, y, z);
```

```
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        moment[0] = v[0];
        moment[1] = v[1];
        r    nent[2] = v[2];
}


void rigid_body::assign_holder1(double x, double y, double z)
{
        holder1[0] = x;
        holder1[1] = y;
        holder1[2] = z;
}


void rigid_body::assign_holder2(double x, double y, double z)
{
        holder2[0] = x;
        holder2[1] = y;
        holder2[2] = z;
}


void rigid_body::assign_holder3(double x, double y, double z, double w)
{
        holder3[0] = x;
        holder3[1] = y;
        holder3[2] = z;
        holder3[3] = w;
}


void rigid_body::assign_velocity_bc(double x, double y, double z)
{
        vector3D v(x, y, z);
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        velocity[0] = v[0];
        velocity[1] = v[1];
        velocity[2] = v[2];
```

```
}


void rigid_body::assign_acceleration_bc(double x, double y, double z)
{
        vector3D v(x, y, z);
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        acceleration[0] = v[0];
        acceleration[1] = v[1];
        acceleration[2] = v[2];
}


void rigid_body::assign_force_bc(double x, double y, double z)
{
        vector3D v(x, y, z);
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        force[0] = v[0];
        force[1] = v[1];
        force[2] = v[2];
}


void rigid_body::assign_ang_velocity_bc(double x, double y, double z)
{
        ang_velocity[0] = x;
        ang_velocity[1] = y;
        ang_velocity[2] = z;
}


void rigid_body::assign_ang_acceleration_bc(double x, double y, double z)
{
        ang_acceleration[0] = x;
        ang_acceleration[1] = y;
        ang_acceleration[2] = z;
}
```

```
void rigid_body::assign_moment_bc(double x, double y, double z)
{
        moment[0] = x;
        moment[1] = y;
        moment[2] = z;
}


void rigid_body::assign_location(vector3D v)
{
        (*location)[0] = v[0];
        (*location)[1] = v[1];
        (*location)[2] = v[2];
}


void rigid_body::assign_velocity(vector3D v)
{
        velocity[0] = v[0];
        velocity[1] = v[1];
        velocity[2] = v[2];
}


void rigid_body::assign_acceleration(vector3D v)
{
        acceleration[0] = v[0];
        acceleration[1] = v[1];
        acceleration[2] = v[2];
}


void rigid_body::assign_force(vector3D v)
{
        force[0] = v[0];
        force[1] = v[1];
        force[2] = v[2];
}


void rigid_body::assign_ang_velocity(vector3D v)
{
        matrix3x3 rotation;
```

```
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        ang_velocity[0] = v[0];
        ang_velocity[1] = v[1];
        ang_velocity[2] = v[2];
}


void rigid_body::assign_ang_acceleration(vector3D v)
{
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        ang_acceleration[0] = v[0];
        ang_acceleration[1] = v[1];
        ang_acceleration[2] = v[2];
}


void rigid_body::assign_moment(vector3D v)
{
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * v;
        moment[0] = v[0];
        moment[1] = v[1];
        moment[2] = v[2];
}


void rigid_body::assign_holder1(vector3D v)
{
        holder1 = v;
}


void rigid_body::assign_holder2(vector3D v)
{
        holder2 = v;
}


void rigid_body::assign_holder3(quaternion v)
```

```
{
        holder3 = v;
}


void rigid_body::assign_velocity_bc(vector3D v)
{
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        velocity[0] = v[0];
        velocity[1] = v[1];
        velocity[2] = v[2];
}


void rigid_body::assign_acceleration_bc(vector3D v)
{
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        acceleration[0] = v[0];
        acceleration[1] = v[1];
        acceleration[2] = v[2];
}


void rigid_body::assign_force_bc(vector3D v)
{
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * v;
        velocity[0] = v[0];
        velocity[1] = v[1];
        velocity[2] = v[2];
}


void rigid_body::assign_ang_velocity_bc(vector3D v)
{
        ang_velocity[0] = v[0];
        ang_velocity[1] = v[1];
        ang_velocity[2] = v[2];
```

```
}


void rigid_body::assign_ang_acceleration_bc(vector3D v)
{
        ang_acceleration[0] = v[0];
        ang_acceleration[1] = v[1];
        ang_acceleration[2] = v[2];
}


void rigid_body::assign_moment_bc(vector3D v)
{
        moment[0] = v[0];
        moment[1] = v[1];
        moment[2] = v[2];
}


int rigid_body::return_type()
{
        return type_body;
}


double rigid_body::return_mass()
{
        return mass;
}


vector3D rigid_body::return_inertia()
{
        vector3D temp;
        temp[0] = inertia[0];
        temp[1] = inertia[4];
        temp[2] = inertia[8];
        return temp;
}


double rigid_body::r urn_surface_area()
{
```

```
        return surface_area;
}


vector3D rigid_body::return_size()
{
        return size;
}


vector3D rigid_body::return_location()
{
        return *location;
}


vector3D* rigid_body::return_location_ptr()
{
        return location;
}


vector3D rigid_body::return_velocity()
{
        return velocity;
}


vector3D rigid_body::return_acceleration()
{
        return acceleration;
}


vector3D rigid_body::return_force()
{
        return force;
}


quaternion rigid_body::return_orientation()
{
        return orientation;
```

```
}


vector3D rigid_body::return_ang_velocity()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * ang_velocity;
        return v;
}


vector3D rigid_body::return_ang_acceleration()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * ang_acceleration;
        return v;
}


vector3D rigid_body::return_moment()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_body_to_world(orientation);
        v = rotation * moment;
        return v;
}


OBJECT* rigid_body::return_shape()
{
        return shape;
}


vector3D rigid_body::return_holder1()
{
        return holder1;
}
```

```
vector3D rigid_body::return_holder2()
{
        return holder2;
}


quaternion rigid_body::return_holder3()
{
        return holder3;
}


vector3D rigid_body::return_velocity_bc()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * velocity;
        return v;
}


vector3D rigid_body::return_acceleration_bc()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * acceleration;
        return v;
}


vector3D rigid_body::return_force_bc()
{
        vector3D v;
        matrix3x3 rotation;
        rotation.DCM_world_to_body(orientation);
        v = rotation * force;
        return v;
}
```

```
vector3D rigid_body::return_ang_velocity_bc()
{
        return ang_velocity;
}


vector3D rigid_body::return_ang_acceleration_bc()
{
        return ang_acceleration;
}


vector3D rigid_body::return_moment_bc()
{
        return moment;
}


void rigid_body::update_state()
{
        vector3D gravity(0.0, -9.81, 0.0);
        matrix3x3 rotation;
        double dt = read_delta();

        if(gravity_status())
        {
                force = force + (gravity * mass);
        }

        if(air_resistance_status())
        {
                double magnitude;
                vector3D direction = velocity * -1.0;
                direction.normalize();
                magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 *
                                surface_area;
                force = force + (direction * magnitude);
        }

        acceleration = force / mass;
        velocity = velocity + (acceleration * dt);
        *location = *location + (velocity * dt) - (acceleration * (0.5 * dt * dt));
```

```
        ang_acceleration[0] = moment[0] / inertia[0];
        ang_acceleration[1] = moment[1] / inertia[4];
        ang_acceleration[2] = moment[2] / inertia[8];
        ang_velocity = ang_velocity + (ang_acceleration * dt);
        orientation.update(ang_velocity, dt);
        orientation.normalize();
        force[0] = 0.0;
        force[1] = 0.0;
        force[2] = 0.0;
        moment[0] = 0.0;
        moment[1] = 0.0;
        moment[2] = 0.0;
}



void rigid_body::update_state_rk4()
{
        double dt = read_delta();
        double hh = dt * .5, h6 = dt / 6;
        vector3D ya = ang_velocity, dyma, dyta, yta, dydxa;
        vector3D yv = velocity, dymv, dytv, ytv, dydxv;
        vector3D yl = *location, dyml, dytl, ytl, dydxl;
        quaternion y = orientation, dym, dyt, yt, dydx; int i;
        vector3D gravity(0.0, -9.81, 0.0);
        matrix3x3 rotation;

        if(gravity_status())
        {
                force = force + (gravity * mass);
        }

        if(air_resistance_status())
        {
                double magnitude;
                vector3D direction = velocity * -1.0;
                direction.normalize();
                magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 *
                                surface_area;
                force = force + (direction * magnitude);
        }

        acceleration = force / mass;
```

```
dydxv = acceleration; dydxl = velocity;
dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] -
            inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] -
            inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] -
            inertia[0]))) / inertia[8];
dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] *
            ang_velocity[1]) + (orientation[3] * ang_velocity[2]));
dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] *
            ang_velocity[2]) - (orientation[3] * ang_velocity[1]));
dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] *
            ang_velocity[0]) - (orientation[1] * ang_velocity[2]));
dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] *
            ang_velocity[1]) - (orientation[2] * ang_velocity[0]));

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + hh * dydx[i];
        yta[i] = ya[i] + hh * dydxa[i];
        ytv[i] = yv[i] + hh * dydxv[i];
        ytl[i] = yl[i] + hh * dydxl[i];
}
yt[3] = y[3] + hh * dydx[3];
dytv = acceleration; dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + hh * dyt[i];
        yta[i] = ya[i] + hh * dyta[i];
        ytv[i] = yv[i] + hh * dytv[i];
        ytl[i] = yl[i] + hh * dytl[i];
}
yt[3] = y[3] + hh * dyt[3];
dymv = acceleration;
dyml = ytv;
```

```
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
        inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
        inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
        inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + dt * dym[i];
        yta[i] = ya[i] + dt * dyma[i];
        ytv[i] = yv[i] + dt * dymv[i];
        ytl[i] = yl[i] + dt * dyml[i];
}
yt[3] = y[3] + dt * dym[3];

for(i = 0; i < 3; i++)
{
        dym[i] = dym[i] +  dyt[i];
        dyma[i] = dyma[i] +  dyta[i];
        dymv[i] = dymv[i] +  dytv[i];
        dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
        orientation[i] = y[i] + h6 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
        ang_velocity[i] = ya[i] + h6 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
        (*location)[i] = yl[i] + h6 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
```

```
            velocity[i] = yv[i] + h6 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
        }
        orientation[3] = y[3] + h6 * (dydx[3] + dyt[3] + 2.0 * dym[3]);
        orientation.normalize();
        force[0] = 0.0;
        force[1] = 0.0;
        force[2] = 0.0;
        moment[0] = 0.0;
        moment[1] = 0.0;
        moment[2] = 0.0;
}


double rigid_body::update_state_rk45(double eps)
{
        double PRGOW = -0.20, PSHRNK = -0.25, FCOR = .06666666,
                dt = read_delta();
        double SAFETY = 0.9, ERRCON = 6.0e-4, xsav = dt, htry = dt;
        double P = ang_velocity[0], Q = ang_velocity[1], R = ang_velocity[2];
        double hh = dt * .5, h6 = dt / 6, h, dt1, hh1, h61, errmax;
        vector3D ya = ang_velocity, dyma, dyta, yta, dydxa, dysava, ysava, ytempa,
                        ytemp2a;
        vector3D yv = velocity, dymv, dytv, ytv, dydxv, dysavv, ysavv, ytempv, ytemp2v;
        vector3D yl = *location, dyml, dytl, ytl, dydxl, dysavl, ysavl, ytempl, ytemp2l;
        quaternion y = orientation, dym, dyt, yt, dydx, dysav, ysav, ytemp, ytemp2; int i;
        vector3D gravity(0.0, -9.81, 0.0);
        matrix3x3 rotation;

        if(gravity_status())
        {
                force = force + (gravity * mass);
        }

        if(air_resistance_status())
        {
                double magnitude;
                vector3D direction = velocity * -1.0;
                direction.normalize();
                magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 *
                                surface_area;
                force = force + (direction * magnitude);
        }
        acceleration = force / mass;
```

```
for(i = 0; i < 3; i++)
{
        ysav[i] = y[i];
        dysav[i] = dydx[i];
        ysava[i] = ya[i];
        dysava[i] = dydxa[i];
        ysavv[i] = yv[i];
        dysavv[i] = dydxv[i];
        ysavl[i] = yl[i];
        dysavl[i] = dydxl[i];
}
ysav[3] = y[3];
dysav[3] = dydx[3];
h = htry;
for(;;)
{
        hh = 0.5 * h;
        dt1 = hh;
        hh1 = dt1 * 0.5;
        h61 = dt1 / 6.0;
        dydxv = acceleration;
        dydxl = velocity;
        dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] -
                inertia[4]))) / inertia[0];
        dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] -
                inertia[8]))) / inertia[4];
        dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] -
                inertia[0]))) / inertia[8];
        dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] *
                ang_velocity[1]) + (orientation[3] * ang_velocity[2]));
        dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] *
                ang_velocity[2]) - (orientation[3] * ang_velocity[1]));
        dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] *
                ang_velocity[0]) - (orientation[1] * ang_velocity[2]));
        dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] *
                ang_velocity[1]) - (orientation[2] * ang_velocity[0]));

        for(i = 0; i < 3; i++)
        {
                yt[i] = ysav[i] + hh1 * dydx[i];
                yta[i] = ysava[i] + hh1 * dydxa[i];
                ytv[i] = ysavv[i] + hh1 * dydxv[i];
```

```
        ytl[i] = ysavl[i] + hh1 * dydxl[i].
}
yt[3] = ysav[3] + hh1 * dydx[3].
dytv = acceleration.
dytl = ytv.
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2])).
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1])).
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2])).
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0])).
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4])))
                        inertia[0].
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8])))
                        inertia[4].
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0])))
                        inertia[8].

for(i = 0, i < 3, i++)
{
        yt[i] = ysav[i] + hh1 * dyt[i].
        yta[i] = ysava[i] + hh1 * dyta[i].
        ytv[i] = ysavv[i] + hh1 * dytv[i].
        ytl[i] = ysavl[i] + hh1 * dytl[i].
}
yt[3] = ysav[3] + hh1 * dyt[3];
dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = ysav[i] + dt1 * dym[i];
        yta[i] = ysava[i] + dt1 * dyma[i];
        ytv[i] = ysavv[i] + dt1 * dymv[i];
        ytl[i] = ysavl[i] + dt1 * dyml[i];
```

```
}
yt[3] = [3] + dt1 * dym[3];

for(i =        ; i++)
{
        dym[i] = dym[i] +  dyt[i];
        dyma[i] = dyma[i] +  dyta[i];
        dymv[i] = dymv[i] +  dytv[i];
        dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) +  [2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) +  * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + ( ] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                        inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                        inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                        inertia[8];

for(i = 0; i < 3; i++)
{
        ytemp[i] = ysav[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
        ytempa[i] = ysava[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
        ytempl[i] = ysavl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
        ytempv[i] = ysavv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
}
ytemp[3] = ysav[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);

//assign new y's for second rk4
for(i = 0; i < 3; i++)
{
        y[i] = ytemp[i];
        ya[i] = ytempa[i];
        yl[i] = ytempl[i];
        yv[i] = ytempv[i];
}
y[3] = ytemp[3];
```

```
//calculate new set of derivatives
dydxv = acceleration;
dydxl = ytempv;
dydx[0] = -0.5*((ytemp[1] * ytempa[0]) + (ytemp[2] * ytempa[1]) +
                (ytemp[3] * ytempa[2]));
dydx[1] = 0.5*((ytemp[0] * ytempa[0]) + (ytemp[2] * ytempa[2]) -
                (ytemp[3] * ytempa[1]));
dydx[2] = 0.5*((ytemp[0] * ytempa[1]) + (ytemp[3] * ytempa[0]) -
                (ytemp[1] * ytempa[2]));
dydx[3] = 0.5*((ytemp[0] * ytempa[2]) + (ytemp[1] * ytempa[1]) -
                (ytemp[2] * ytempa[0]));
dydxa[0] = (moment[0] - (ytempa[1] * ytempa[2] * (inertia[8] -
                inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ytempa[0] * ytempa[2] * (inertia[0] -
                inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ytempa[1] * ytempa[0] * (inertia[4] -
                inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + hh1 * dydx[i];
        yta[i] = ya[i] + hh1 * dydxa[i];
        ytv[i] = yv[i] + hh1 * dydxv[i];
        ytl[i] = yl[i] + hh1 * dydxl[i];
}
yt[3] = y[3] + hh1 * dydx[3];
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + hh1 * dyt[i];
        yta[i] = ya[i] + hh1 * dyta[i];
```

```
        ytv[i] = yv[i] + hh1 * dytv[i];
        ytl[i] = yl[i] + hh1 * dytl[i];
}
yt[3] = y[3] + hh1 * dyt[3];
dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = y[i] + dt1 * dym[i];
        yta[i] = ya[i] + dt1 * dyma[i];
        ytv[i] = yv[i] + dt1 * dymv[i];
        ytl[i] = yl[i] + dt1 * dyml[i];
}
yt[3] = y[3] + dt1 * dym[3];

for(i = 0; i < 3; i++)
{
        dym[i] = dym[i] +  dyt[i];
        dyma[i] = dyma[i] +  dyta[i];
        dymv[i] = dymv[i] +  dytv[i];
        dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                        inertia[0];
```

```
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                        inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                        inertia[8];

for(i = 0; i < 3; i++)
{
        ytemp2[i] = y[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
        ytemp2a[i] = ya[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
        ytemp2l[i] = yl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
        ytemp2v[i] = yv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
}
ytemp2[3] = y[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);

//third rk4 run
dt1 = h;
hh1 = dt1 * 0.5;
h51 = dt1 / 6.0;
dydxv = acceleration;
dydxl = velocity;
dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] -
                        inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] -
                        inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] -
                        inertia[0]))) / inertia[8];
dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] *
                ang_velocity[1]) + (orientation[3] * ang_velocity[2]));
dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] *
                ang_velocity[2]) - (orientation[3] * ang_velocity[1]));
dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] *
                ang_velocity[0]) - (orientation[1] * ang_velocity[2]));
dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] *
                ang_velocity[1]) - (orientation[2] * ang_velocity[0]));

for(i = 0; i < 3; i++)
{
        yt[i] = ysav[i] + hh1 * dydx[i];
        yta[i] = ysava[i] + hh1 * dydxa[i];
        ytv[i] = ysavv[i] + hh1 * dydxv[i];
        ytl[i] = ysavl[i] + hh1 * dydxl[i];
}
yt[3] = ysav[3] + hh1 * dydx[3];
```

```
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                        inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                        inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                        inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = ysav[i] + hh1 * dyt[i];
        yta[i] = ysava[i] + hh1 * dyta[i];
        ytv[i] = ysavv[i] + hh1 * dytv[i];
        ytl[i] = ysavl[i] + hh1 * dytl[i];
}
yt[3] = ysav[3] + hh1 * dyt[3];
dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                inertia[8];

for(i = 0; i < 3; i++)
{
        yt[i] = ysav[i] + dt1 * dym[i];
        yta[i] = ysava[i] + dt1 * dyma[i];
        ytv[i] = ysavv[i] + dt1 * dymv[i];
        ytl[i] = ysavl[i] + dt1 * dyml[i];
}
yt[3] = ysav[3] + dt1 * dym[3];
```

```
for(i = 0; i < 3; i++)
{
        dym[i] = dym[i] +  dyt[i];
        dyma[i] = dyma[i] +  dyta[i];
        dymv[i] = dymv[i] +  dytv[i];
        dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) /
                        inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) /
                        inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) /
                        inertia[8];

for(i = 0; i < 3; i++)
{
        ytemp[i] = ysav[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
        ytempa[i] = ysava[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
        ytempl[i] = ysavl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
        ytempv[i] = ysavv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
}
ytemp[3] = ysav[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);

//error determination
errmax = 0.0;
for(i = 0; i < 3; i++)
{
        ytemp[i] = ytemp2[i] - ytemp[i];

        if(errmax < fabs(ytemp[i]))
                errmax = fabs(ytemp[i]);
        ytempa[i] = ytemp2a[i] - ytempa[i];

        if(errmax < fabs(ytempa[i]))
                errmax = fabs(ytempa[i]);
        ytempl[i] = ytemp2l[i] - ytempl[i];
```

```
                    if(errmax < fabs(ytempl[i]))
                            errmax = fabs(ytempl[i]);
                    ytempv[i] = ytemp2v[i] - ytempv[i];

                    if(errmax < fabs(ytempv[i]))
                            errmax = fabs(ytempv[i]);
            }
            ytemp[3] = ytemp2[3] - ytemp[3];

            if(errmax < fabs(ytemp[3]))
                    errmax = fabs(ytemp[3]);
            errmax /= eps;

            if(errmax < 1.0)
                    break;
            h = SAFETY * h * exp(PSHRNK*log(errmax));
        }

        for(i = 0; i < 3; i++)
        {
        orientation[i] = ytemp2[i] + ytemp[i] * FCOR;
        ang_velocity[i] = ytemp2a[i] + ytempa[i] * FCOR;
        (*location)[i] = ytemp2l[i] + ytempl[i] * FCOR;
        velocity[i] = ytemp2v[i] + ytempv[i] * FCOR;
        }
        orientation[3] = ytemp2[3] + ytemp[3] * FCOR;
        orientation.normalize();
        force[0] = 0.0;
        force[1] = 0.0;
        force[2] = 0.0;
        moment[0] = 0.0;
        moment[1] = 0.0;
        moment[2] = 0.0;
        return h;
}


void rigid_body::display()
{
        Matrix rt = {   1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, 1.0, 0.0,
```

```
                    0.0, 0.0, 0.0, 1.0};

        Matrix scale = { 1.0, 0.0, 0.0, 0.0,
                         0.0, 1.0, 0.0, 0.0,
                         0.0, 0.0, 1.0, 0.0,
                         0.0, 0.0, 0.0, 1.0};

        pushmatrix();
        rt[0][0] = ((orientation[0] * orientation[0]) + (orientation[1] *  orientation[1]) -
                    (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        rt[1][0] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] *
orientation[3]));        rt[2][0] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1]
*                                                   orientation[3]));
        rt[0][1] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] *
                          orientation[3]));
        rt[1][1] = ((orientation[0] * orientation[0]) - (orientation[1] * orientation[1]) +
                    (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        rt[2][1] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] *
orientation[1]));
        rt[0][2] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] *
orientation[2]));
        rt[1][2] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] *
                          orientation[1]));
        rt[2][2] = ((orientation[0] * orientation[0]) - (orientation[1] * orientation[1]) -
                    (orientation[2] * orientation[2]) + (orientation[3] * orientation[3]));
        rt[3][0] = (*location)[0];
        rt[3][1] = (*location)[1];
        rt[3][2] = (*location)[2];
        multmatrix(rt);
        scale[0][0] = size[0];
        scale[1][1] = size[1];
        scale[2][2] = size[2];
        multmatrix(scale);

        if(display_axis && *display_shape)
        {
                view_axis();
        }

        if(*display_shape)
        {
                display_this_object(shape);
        }
```

```
        popmatrix();
        }


void rigid_body::add_axis()
{
        display_axis = 1;
}


void rigid_body::remove_axis()
{
        display_axis = 0;
}


void rigid_body::attach_eye()
{
        attach_eye_to(location, display_shape);
}


void rigid_body::attach_target()
{
        attach_target_to(location);
}


void set_eye(double x, double y, double z)
{
        set_eye_to(x, y, z);
}


void set_target(double x, double y, double z)
{
        set_target_to(x, y, z);
}


void rigid_body::zero()
{
        vector3D zero_vector;
```

```
        size[0] = 1.0;
        size[1] = 1.0;
        size[2] = 1.0;
        *location = zero_vector;
        velocity = zero_vector;
        acceleration = zero_vector;
        orientation[0] = 1.0;
        orientation[1] = 0.0;
        orientation[2] = 0.0;
        orientation[3] = 0.0;
        ang_velocity = zero_vector;
        ang_acceleration = zero_vector;
}



void rigid_body::attached_body_update(rigid_body r)
{
        vector3D av;
        matrix3x3 rotation;
        *location = holder1;
        update_state();
        holder1 = *location;
        rotation.DCM_body_to_world(r.orientation);
        *location = (rotation * (*location)) + *r.location;
        holder2 = (rotation * r.ang_velocity) + r.holder2;
        av = holder2;
        rotation.DCM_world_to_body(orientation);
        av = rotation * av;
        orientation.update(av,read_delta());
}

#endif
```

APPENDIX D. VECTOR3D CODE

A. HEADER FILE

```
#ifndef VECTOR3D_H
#define VECTOR3D_H
#include <iostream.h>
#include <math.h>

class vector3D
{
        double x;
        double y;
        double z; public:
        vector3D();
        vector3D(double, double, double);
        vector3D(const vector3D&);
        void zero();
        vector3D& operator=(const vector3D&);
        vector3D operator+(const vector3D&);
        vector3D operator-(const vector3D&);
        double operator*(const vector3D&);              //dot product
        vector3D operator*(double);                     //scalar multiplication
        vector3D operator/(double);                 //scalar division
        vector3D operator^(const vector3D&);        //cross product
        double magnitude();
        void normalize();
        void normalize(double);
        friend  ostream& operator<<(ostream&, vector3D&);
        double& operator[](int);
        ~vector3D() { }
};

#endif
```

B. SOURCE FILE

```
#ifndef VECTOR3D_C
#define VECTOR3D_C
#include "vector3D.H"

//default constructor vector3D::vector3D()
{
```

```
        x = 0.0;
        y = 0.0;
        z = 0.0;
}


//constructor using three doubles
vector3D::vector3D(double a, double b, double c)
{
        x = a;
        y = b;
        z = c;
}


//constructor using another vector3D
vector3D::vector3D(const vector3D& v)
{
        x = v.x;
        y = v.y;
        z = v.z;
}


//zero out the vector
void vector3D::zero()
{
        x = 0.0;
        y = 0.0;
        z = 0.0;
}


//Assignment operator - the function must return a reference to a vector
//instead of a vector for assignment to work properly
 vector3D& vector3D::operator=(const vector3D& v)
{
        x = v.x;
        y = v.y;
        z = v.z;
        return *this;
}
```

```
//vector addition operator
vector3D vector3D  operator+(const vector3D& v)
{
        vector3D sum;
        sum.x = v.x + x;
        sum.y = v.y + y;
        sum.z = v.z + z;
        return sum;
}


//vector substraction
vector3D vector3D::operator-(const vector3D& v)
{
        vector3D diff;
        diff.x = x - v.x;
        diff.y = y - v.y;
        diff.z = z - v.z;
        return diff;
}


//vector dot product
double vector3D::operator*(const vector3D& v)
{
        double dot;
        dot = (v.x * x) + (v.y * y) + (v.z * z);
        return dot;
}


//scalar multiplication vector3D vector3D::operator*(double n)
{
        vector3D mult;
        mult.x = x * n;
        mult.y = y * n;
        mult.z = z * n;
        return mult;
}
```

```
 scalar division - it is the user responsibility to make sure that n is not zero vector3D
vector3D  operator/(double n)
{
        vector3D result;
        result.x = x / n;
        result.y = y / n;
        result.z = z / n;
        return result;
}


//vector cross product
vector3D vector3D::operator^(const vector3D& v)
{
        vector3D cross;
        cross.x = (y * v.z) - (v.y * z);
        cross.y = -((x * v.z) - (v.x * z));
        cross.z = (x * v.y) - (v.x * y);
        return cross;
}


//the << operator is to be used with cout
ostream& operator<<(ostream& os, vector3D& v)
{
        os << (double) v.x << ", " << (double) v.y << ", " << (double) v.z << "\n";
        return os;
}


//allows access to the components of the vector3D.  it must return a reference
//in order for assignment to work
double& vector3D::operator[](int n)
{
        if (n == 0)
        {
                return x;
        }
        if (n == 1)
        {
                return y;
        }
        if (n == 2)
```

```
        {
                return z;
        }
}


//returns the magnitude of the vector
double vector3D::magnitude()
{
        return sqrt((x * x) + (y * y) + (z * z));
}


//normalizes the vector to one
void vector3D::normalize()
{
        double m = sqrt((x * x) + (y * y) + (z * z));
        if (m)
        {
                x = x / m;
                y = y / m;
                z = z / m;
        }
}


//normalize the vector to d
void vector3D::normalize(double d)
{
        double m = sqrt((x * x) + (y * y) + (z * z));
        if (m)
        {
                x = d * x / m;
                y = d * y / m;
                z = d * z / m;
        }
}


#endif
```

APPENDIX E. MATRIX3X3 CODE

A. HEADER FILE

```
#ifndef MATRIX3X3_H
#define MATRIX3X3_H
#include "vector3D.H"
#include "quaternion.H"
#include <iostream.h>

class matrix3x3
{
        double m[9];
public:
        matrix3x3();
        matrix3x3(double, double, double, double, double, double, double, double,
                        double);
        matrix3x3(const matrix3x3&);
        void reset();
        matrix3x3& operator=(const matrix3x3&);
        matrix3x3 operator+(const matrix3x3&);
        matrix3x3 operator-(const matrix3x3&);
        matrix3x3 operator*(const matrix3x3&);
        void DCM_x_rotation(double);
        void DCM_y_rotation(double);
        void DCM_z_rotation(double);
        void DCM_body_to_world(quaternion);
        void DCM_world_to_body(quaternion);
        void transpose();
        quaternion generate_orientation();
        vector3D operator*(vector3D&);
        matrix3x3 operator*(double);
        double& operator[](int);
        friend  ostream& operator<<(ostream&, matrix3x3&);
        ~matrix3x3() { }
};

#endif
```

B. SOURCE FILE

```
#ifndef MATRIX3X3_C
#define MATRIX3X3_C
```

```
#include "matrix3x3.H"

matrix3x3::matrix3x3()
{
        m[0] = 1;
        m[1] = 0;
        m[2] = 0;
        m[3] = 0;
        m[4] = 1;
        m[5] = 0;
        m[6] = 0;
        m[7] = 0;
        m[8] = 1;
}


matrix3x3::matrix3x3(double a, double b, double c, double d, double e, double f, double g,
                        double h, double i)
{
        m[0] = a;
        m[1] = b;
        m[2] = c;
        m[3] = d;
        m[4] = e;
        m[5] = f;
        m[6] = g;
        m[7] = h;
        m[8] = i;
}


matrix3x3::matrix3x3(const matrix3x3& v)
{
        m[0] = v.m[0];
        m[1] = v.m[1];
        m[2] = v.m[2];
        m[3] = v.m[3];
        m[4] = v.m[4];
        m[5] = v.m[5];
        m[6] = v.m[6];
        m[7] = v.m[7];
        m[8] = v.m[8];
}
```

```
void matrix3x3::reset()
{
        m[0] = 1;
        m[1] = 0;
        m[2] = 0;
        m[3] = 0;
        m[4] = 1;
        m[5] = 0;
        m[6] = 0;
        m[7] = 0;
        m[8] = 1;
}


matrix3x3& matrix3x3::operator=(const matrix3x3& v)
{
        m[0] = v.m[0];
        m[1] = v.m[1];
        m[2] = v.m[2];
        m[3] = v.m[3];
        m[4] = v.m[4];
        m[5] = v.m[5];
        m[6] = v.m[6];
        m[7] = v.m[7];
        m[8] = v.m[8];
        return *this;
}


matrix3x3 matrix3x3::operator+(const matrix3x3& v)
{
        matrix3x3 sum;
        sum.m[0] = m[0] + v.m[0];
        sum.m[1] = m[1] + v.m[1];
        sum.m[2] = m[2] + v.m[2];
        sum.m[3] = m[3] + v.m[3];
        sum.m[4] = m[4] + v.m[4];
        sum.m[5] = m[5] + v.m[5];
        sum.m[6] = m[6] + v.m[6];
        sum.m[7] = m[7] + v.m[7];
        sum.m[8] = m[8] + v.m[8];
```

```
        return sum;
}


matrix3x3 matrix3x3::operator-(const matrix3x3& v)
{
        matrix3x3 difference;
        difference.m[0] = m[0] - v.m[0];
        difference.m[1] = m[1] - v.m[1];
        difference.m[2] = m[2] - v.m[2];
        difference.m[3] = m[3] - v.m[3];
        difference.m[4] = m[4] - v.m[4];
        difference.m[5] = m[5] - v.m[5];
        difference.m[6] = m[6] - v.m[6];
        difference.m[7] = m[7] - v.m[7];
        difference.m[8] = m[8] - v.m[8];
        return difference;
}


matrix3x3 matrix3x3::operator*(const matrix3x3& v)
{
        matrix3x3 temp;
        temp.m[0] = (m[0] * v.m[0]) + (m[1] * v.m[3]) + (m[2] * v.m[6]);
        temp.m[1] = (m[0] * v.m[1]) + (m[1] * v.m[4]) + (m[2] * v.m[7]);
        temp.m[2] = (m[0] * v.m[2]) + (m[1] * v.m[5]) + (m[2] * v.m[8]);
        temp.m[3] = (m[3] * v.m[0]) + (m[4] * v.m[3]) + (m[5] * v.m[6]);
        temp.m[4] = (m[3] * v.m[1]) + (m[4] * v.m[4]) + (m[5] * v.m[7]);
        temp.m[5] = (m[3] * v.m[2]) + (m[4] * v.m[5]) + (m[5] * v.m[8]);
        temp.m[6] = (m[6] * v.m[0]) + (m[7] * v.m[3]) + (m[8] * v.m[6]);
        temp.m[7] = (m[6] * v.m[1]) + (m[7] * v.m[4]) + (m[8] * v.m[7]);
        temp.m[8] = (m[6] * v.m[2]) + (m[7] * v.m[5]) + (m[8] * v.m[8]);
        return temp;
}


vector3D matrix3x3::operator*(vector3D& v)
{
        vector3D temp = v;
        temp[0] = (m[0] * v[0]) + (m[1] * v[1]) + (m[2] * v[2]);
        temp[1] = (m[3] * v[0]) + (m[4] * v[1]) + (m[5] * v[2]);
        temp[2] = (m[6] * v[0]) + (m[7] * v[1]) + (m[8] * v[2]);
        return temp;
```

```
}


        matrix3x3 matrix3x3::operator*(double n)
{
        matrix3x3 product;
        product.m[0] = m[0] * n;
        product.m[1] = m[1] * n;
        product.m[2] = m[2] * n;
        product.m[3] = m[3] * n;
        product.m[4] = m[4] * n;
        product.m[5] = m[5] * n;
        product.m[6] = m[6] * n;
        product.m[7] = m[7] * n;
        product.m[8] = m[8] * n;
        return product;
}


ostream& operator<<(ostream& os, matrix3x3& v)
{
        os << (double) v.m[0] << ", " << (double) v.m[1] << ", " << (double)v.m[2]
                <<"\n"
           << (double) v.m[3] << ", " << (double) v.m[4] << ", " << (double) v.m[5]
                << "\n"
          << (double) v.m[6] << ", " << (double) v.m[7] << ", " << (double) v.m[8]
                << "\n" << "\n";
        return os;
}


double& matrix3x3::operator[](int y)
{
        return m[y];
}


void matrix3x3::DCM_x_rotation(double angle)
{
        m[0] = 1.0;
        m[1] = 0.0;
        m[2] = 0.0;
        m[3] = 0.0;
```

```
        m[4] = cos(angle);
        m[5] = sin(angle);
        m[6] = 0.0;
        m[7] = -sin(angle);
        m[8] = cos(angle);
}


void matrix3x3::DCM_y_rotation(double angle)
{
        m[0] = cos(angle);
        m[1] = 0.0;
        m[2] = -sin(angle);
        m[3] = 0.0;
        m[4] = 1.0;
        m[5] = 0.0;
        m[6] = sin(angle);
        m[7] = 0.0;
        m[8] = cos(angle);
}


void matrix3x3::DCM_z_rotation(double angle)
{
        m[0] = cos(angle);
        m[1] = sin(angle);
        m[2] = 0.0;
        m[3] = -sin(angle);
        m[4] = cos(angle);
        m[5] = 0.0;
        m[6] = 0.0;
        m[7] = 0.0;
        m[8] = 1.0;
}


void matrix3x3::DCM_body_to_world(quaternion orientation)
{
        m[0] = ((orientation[0] * orientation[0]) + (orientation[1] * orientation[1]) -
                (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        m[1] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] * orientation[3]));
        m[2] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1] * orientation[3]));
        m[3] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] * orientation[3]));
```

```
        m[4] = ((orientation[0] * orientation[0]) - (orientation[1] * orientation[1]) +
                (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        m[5] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] * orientation[1]));
        m[6] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] * orientation[2]));
        m[7] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] * orientation[1]));
        m[8] = ((orientation[0] * orientation[0]) - (orientation[1] *  orientation[1]) -
                (orientation[2] * orientation[2]) + (orientation[3] * orientation[3]));
}


void matrix3x3::DCM_world_to_body(quaternion orientation)
{
        m[0] = ((orientation[0] * orientation[0]) + (orientation[1] * orientation[1]) -
                (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        m[3] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] * orientation[3]));
        m[6] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1] * orientation[3]));
        m[1] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] * orientation[3]));
        m[4] = ((orientation[0] * orientation[0]) - (orientation[1] *  orientation[1]) +
                (orientation[2] * orientation[2]) - (orientation[3] * orientation[3]));
        m[7] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] *orientation[1]));
        m[2] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] * orientation[2]));
        m[5] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] * orientation[1]));
        m[8] = ((orientation[0] * orientation[0]) - (orientation[1] * orientation[1]) -
                        (orientation[2] * orientation[2]) + (orientation[3] *
orientation[3]));
}


void matrix3x3::transpose()
{
        matrix3x3 v = *this;
        m[0] = v.m[0];
        m[1] = v.m[3];
        m[2] = v.m[6];
        m[3] = v.m[1];
        m[4] = v.m[4];
        m[5] = v.m[7];
        m[6] = v.m[2];
        m[7] = v.m[5];
        m[8] = v.m[8];
}
```

```
quaternion matrix3x3::generate_orientation()
{
        quaternion q;
        q[0] = 0.5 * sqrt(1 + m[0] + m[4] + m[8]);
        q[1] = 0.5 * sqrt(1 + m[0] - m[4] - m[8]);
        q[2] = 0.5 * sqrt(1 - m[0] + m[4] - m[8]);
        q[3] = 0.5 * sqrt(1 - m[0] - m[4] + m[8]);
        q.normalize();
        return q;
}


#endif
```

APPENDIX F. QUATERNION CODE

A. HEADER FILE

```
#ifndef QUATERNION_H
#define QUATERNION_H
#include <iostream.h>
#include <math.h>
#include "vector3D.H"

class quaternion
{
        double q0;
        double q1;
        double q2;
        double q3;

public:
        quaternion();
        quaternion(double, double, double, double);
        quatern          ion(const quaternion&);
        void set(double, double, double, double);
        quaternion& operator=(const quaternion&);
        quaternion operator+(const quaternion&);
        quaternion operator-(const quaternion&);
        quaternion operator*(const quaternion&);
        quaternion operator*(double);
        double& operator[](int);
        double magnitude();
        void normalize();
        quaternion rate_of_change(double, double, double);
        void update(double, double, double, double);
        quaternion rate_of_change(vector3D);
        void update(vector3D, double);
        friend ostream& operator<<(ostream&, quaternion&);
        ~quaternion() { }
};

#endif
```

B. SOURCE FILE

```
#ifndef QUATERNION_C
```

```
#define QUATERNION_C
#include "quaternion.H"

quaternion::quaternion()
{
        q0 = 1.0;
        q1 = 0.0;
        q2 = 0.0;
        q3 = 0.0;
}


quaternion::quaternion(double angle_x, double angle_y, double angle_z, double rotation)
{
        q0 = cosf(0.5*rotation);
        q1 = cosf(angle_x)*sinf(0.5*rotation);
        q2 = cosf(angle_y)*sinf(0.5*rotation);
        q3 = cosf(angle_z)*sinf(0.5*rotation);
}


quaternion::quaternion(const quaternion& q)
{
        q0 = q.q0;
        q1 = q.q1;
        q2 = q.q2;
        q3 = q.q3;
}


void quaternion::set(double angle_x, double angle_y, double angle_z, double rotation)
{
        q0 = cosf(0.5*rotation);
        q1 = cosf(angle_x)*sinf(0.5*rotation);
        q2 = cosf(angle_y)*sinf(0.5*rotation);
        q3 = cosf(angle_z)*sinf(0.5*rotation);
}


quaternion& quaternion::operator=(const quaternion& q)
{
        q0 = q.q0;
        q1 = q.q1;
```

```
        q2 = q.q2;
        q3 = q.q3;
        return *this;
}


quaternion quaternion::operator+(const quaternion& q)
{
        quaternion sum;
        sum.q0 = q0 + q.q0;
        sum.q1 = q1 + q.q1;
        sum.q2 = q2 + q.q2;
        sum.q3 = q3 + q.q3;
        return sum;
}


quaternion quaternion::operator-(const quaternion& q)
{
        quaternion diff;
        diff.q0 = q0 - q.q0;
        diff.q1 = q1 - q.q1;
        diff.q2 = q2 - q.q2;
        diff.q3 = q3 - q.q3;
        return diff;
}


quaternion quaternion::operator*(const quaternion& q)
{
        quaternion prod;
        prod.q0 = (q0 * q.q0) - (q1 * q.q1) - (q2 * q.q2) - (q3 * q.q3);
        prod.q1 = (q1 * q.q0) + (q0 * q.q1) - (q3 * q.q2) + (q2 * q.q3);
        prod.q2 = (q2 * q.q0) + (q3 * q.q1) - (q0 * q.q2) + (q1 * q.q3);
        prod.q3 = (q3 * q.q0) + (q2 * q.q1) - (q1 * q.q2) + (q0 * q.q3);
        return prod;
}


quaternion quaternion::operator*(double n)
{
        quaternion prod;
        prod.q0 = q0 * n;
```

```
        prod.q1 = q1 * n;
        prod.q2 = q2 * n;
        prod.q3 = q3 * n;
        return prod;
}


double quaternion::magnitude()
{
        return sqrt((q0 * q0) + (q1 * q1) + (q2 * q2) + (q3 * q3));
}


void quaternion::normalize()
{
        double m = sqrt((q0 * q0) + (q1 * q1) + (q2 * q2) + (q3 * q3));
        if (m)
        {
                q0 = q0 / m;
        q1 = q1 / m;
        q2 = q2 / m;
        q3 = q3 / m;
        }
}


quaternion quaternion::rate_of_change(double P, double Q, double R)
{
        quaternion rate;
        rate.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
        rate.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
        rate.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
        rate.q3 = 0.5*((q0 * R) + (q1 * Q) -  (q2 * P));
        return rate;
}


void quaternion::update(double P, double Q, double R, double sec)
{
        double hh = sec * .5, h6 = sec / 6;
        quaternion y = *this, dym, dyt, yt, dydx;
        dydx.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
        dydx.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
```

```
        dydx.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
        dydx.q3 = 0.5*((q0 * R) + (q1 * Q) - (q2 * P));
        yt.q0 = y.q0 + hh * dydx.q0;
        yt.q1 = y.q1 + hh * dydx.q1;
        yt.q2 = y.q2 + hh * dydx.q2;
        yt.q3 = y.q3 + hh * dydx.q3;
        dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
        dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));
        yt.q0 = y.q0 + hh * dyt.q0;
        yt.q1 = y.q1 + hh * dyt.q1;
        yt.q2 = y.q2 + hh * dyt.q2;
        yt.q3 = y.q3 + hh * dyt.q3;
        dym.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
        dym.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dym.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dym.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));
        yt.q0 = y.q0 + sec * dym.q0;
        yt.q1 = y.q1 + sec * dym.q1;
        yt.q2 = y.q2 + sec * dym.q2;
        yt.q3 = y.q3 + sec * dym.q3;
        dym.q0 = dym.q0 + dyt.q0;
        dym.q1 = dym.q1 + dyt.q1;
        dym.q2 = dym.q2 + dyt.q2;
        dym.q3 = dym.q3 + dyt.q3;
        dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
        dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));
        q0 = y.q0 + h6 * (dydx.q0 + dyt.q0 + 2.0 * dym.q0);
        q1 = y.q1 + h6 * (dydx.q1 + dyt.q1 + 2.0 * dym.q1);
        q2 = y.q2 + h6 * (dydx.q2 + dyt.q2 + 2.0 * dym.q2);
        q3 = y.q3 + h6 * (dydx.q3 + dyt.q3 + 2.0 * dym.q3);
}


quaternion quaternion::rate_of_change(vector3D ang_velocity)
{

        quaternion rate;
        rate.q0 = -0.5*((q1 * ang_velocity[0]) + (q2 * ang_velocity[1]) + (q3 *
ang_velocity[2]));
```

```
        rate.q1 =  0.5*((q0 * ang_velocity[0]) + (q2 * ang_velocity[2]) - (q3 *
ang_velocity[1]));
        rate.q2 =  0.5*((q0 * arg_velocity[1]) + (q3 * ang_velocity[0]) - (q1 *
ang_velocity[2]));              rate.q3 =  0.5*((q0 * ang_velocity[2]) + (q1 *
ang_velocity[1]) - (q2 * ang_velocity[0]));
        return rate;
}


void quaternion::update(vector3D ang_velocity, double sec)
{
        double P = ang_velocity[0], Q = ang_velocity[1], R = ang_velocity[2];
        double hh = sec * .5, h6 = sec / 6;
        quaternion y = *this, dym, dyt, yt, dydx;
        dydx.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
        dydx.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
        dydx.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
        dydx.q3 = 0.5*((q0 * R) + (q1 * Q) - (q2 * P));
        yt.q0 = y.q0 + hh * dydx.q0;
        yt.q1 = y.q1 + hh * dydx.q1;
        yt.q2 = y.q2 + hh * dydx.q2;
        yt.q3 = y.q3 + hh * dydx.q3;
        dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
        dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));
        yt.q0 = y.q0 + hh * dyt.q0;
        yt.q1 = y.q1 + hh * dyt.q1;
        yt.q2 = y.q2 + hh * dyt.q2;
        yt.q3 = y.q3 + hh * dyt.q3;
        dym.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
        dym.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dym.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dym.q3 = 0.5*((yt.q0 * R) +(yt.q1 * Q) - (yt.q2 * P));
        yt.q0 = y.q0 + sec * dym.q0;
        yt.q1 = y.q1 + sec * dym.q1;
        yt.q2 = y.q2 + sec * dym.q2;
        yt.q3 = y.q3 + sec * dym.q3;
        dym.q0 = dym.q0 + dyt.q0;
        dym.q1 = dym.q1 + dyt.q1;
        dym.q2 = dym.q2 + dyt.q2;
        dym.q3 = dym.q3 + dyt.q3;
        dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
```

```
        dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
        dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
        dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));
        q0 = y.q0 + h6 * (dydx.q0 + dyt.q0 + 2.0 * dym.q0);
        q1 = y.q1 + h6 * (dydx.q1 + dyt.q1 + 2.0 * dym.q1);
        q2 = y.q2 + h6 * (dydx.q2 + dyt.q2 + 2.0 * dym.q2);
        q3 = y.q3 + h6 * (dydx.q3 + dyt.q3 + 2.0 * dym.q3);
}


ostream& operator<<(ostream& os, quaternion& q)
{
        os << (double) q.q0 << ", " << (double) q.q1 << ", " << (double) q.q2 << ", "
           << (double) q.q3 << "\n";
        return os;
}


double& quaternion::operator[](int n)
{
        if (n == 0)
        {
                return q0;
        }
        if (n == 1)
        {
                return q1;
        }
        if (n == 2)
        {
                return q2;
        }
        if (n == 3)
        {
                return q3;
        }
}

#endif
```

APPENDIX G. MENU CODE

A. HEADER FILE

```
#ifndef MENU_H
#define MENU_H
#include "menu.H"
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <stdlib.h>

void initialize_menu();
int queue_test();

#endif
```

B. SOURCE FILE

```
#ifndef MENU_C
#define MENU_C
#include "menu.H"
#include "time.H"
#include <iostream.h>

long mainmenu;
long hititem;
short value;
double wx, wy;


long makethemenus()          /* this routine performs all the menu construction calls */
{
        long topmenu;
        topmenu = defpup("Dynamics Visualizer %t | Exit %x99");
        return(topmenu);
}


void initialize_menu()
{
        mainmenu = makethemenus();
}
```

```
void processmenuhit(long hititem)
{
        switch(hititem)
        {
                case 99:
                        exit(0)
                        break;
                default:
                        break;
        }       /* end switch */
}


int queue_test()
{
        hititem = 0;
        while(qtest())
        {
                switch(qread(&value))
                {
                        case MENUBUTTON:
                                if(value == 1)
                                {
                                        mmode(MSINGLE);
                                        hititem = dopup(mainmenu);
                                        mmode(MVIEWING);
                                        processmenuhit(hititem);
                                        reset_time();
                                }
                                break;

                        case LEFTMOUSE:
                                wx = getvaluator(MOUSEX);
                                wy = getvaluator(MOUSEY);
                                hititem = (wx * 100000) + wy;
                        break;

                        case REDRAW:
                                reshapeviewport();
                                break;
```

```
case UPARROWKEY:
        hititem = 100;
        break;

case DOWNARROWKEY:
        hititem = 101;
        break;

case LEFTARROWKEY:
        hititem = 102;
        break;

case RIGHTARROWKEY:
        hititem = 103;
        break;

case EQUALKEY:
        hititem = 104;
        break;

case MINUSKEY:
        hititem = 105;
        break;

case SPACEKEY:
        hititem = 106;
        break;

case F1KEY:
        hititem = 111;
        break;

case F2KEY:
        hititem = 112;
        break;

case F3KEY:
        hititem = 113;
        break;

case F4KEY:
        hititem = 114;
        break;
```

```
                case F5KEY:
                        hititem = 115;
                        break;

                case F6KEY:
                        hititem = 116;
                        break;

                case F7KEY:
                        hititem = 117;
                        break;

                case F8KEY:
                        hititem = 118;
                        break;

                case F9KEY:
                        hititem = 119;
                        break;

                case F10KEY:
                        hititem = 120;
                        break;

                case F11KEY:
                        hititem = 121;
                        break;

                case F12KEY:
                        hititem = 122;
                        break;

                default:
                        break;
            }
        }
        return (int) hititem;
}


#endif
```

APPENDIX H. TIME CODE

A. HEADER FILE

```
#ifndef TIME_H
#define TIME_H
#include <time.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>

void set_delta();
void set_delta(double);
void set_time();
void reset_time();
double read_delta();
double read_time();
int read_ticks();
void set_real_time_factor(double);

#endif
```

B. SOURCE FILE

```
#ifndef TIME_C
#define TIME_C
#include "time.H"

struct tms timebuffer;
long old_time; double delta, real_time_ratio = 1.0;

void set_delta()
{
        delta = ((double) (times(&timebuffer) - old_time)/HZ) * real_time_ratio;
        old_time = times(&timebuffer);
}


void set_delta(double step)
{
        delta = step;
        old_time = times(&timebuffer);
}
```

```
void set_time()
{
        old_time = times(&timebuffer);
}


double read_delta()
{
        return delta;
}


double read_time()
{
        return (double) old_time;
}


int read_ticks()
{
        return (int) times(&timebuffer);
}


void set_real_time_factor(double f)
{
        real_time_ratio = f;
}


void reset_time()
{
        long delta_ticks;
        delta_ticks = (long) (delta * HZ);
        old_time = times(&timebuffer) - delta_ticks;
}

#endif
```

APPENDIX I. ALTERING THE GRAVITY GRADIENT VISUALIZER CODE

Altering the Gravity Gradient Visualizer code is a tedious process not to be attempted unless one has a working knowledge of the UNIX operating system, computer graphics and C++ programming. If the user feels the need to make changes to the code, the following steps should be followed:

- After logging on to a Silicon Graphics computer, at the UNIX prompt type "cd ~jstewart/thesis". This will put the user in the directory where the code resides.
- Using any text editor, edit the file to be changed, make and save the changes, and exit the editor.
- At the UNIX prompt type "make". This will run a makefile which will determine that there is an update to a file and recompile and relink the file.
- Run the program as usual.

As an example, to change the default click value for the mass field from 50 kilogram increments to 10 kilogram increments, log on, change to the proper directory and edit the file "ggrad.C". Page down the file until you find a line that reads "mass = mass + 50;" and change the "50" to "10". Then find a line that reads "mass = mass - 50;" and change the "50" to "10". Save the changes and exit the editor. At the UNIX prompt type "make" to recompile and relink the code. Run the program as usual. This process can be repeated for any changes one wishes to make. Although this is a straightforward process, there are many opportunities for error, and these errors can be difficult to find. Therefore, it can

not be emphasized enough that this should not be attempted by one without the requisite background in the aforementioned areas.

LIST OF REFERENCES

1. Mitchell & Gauthier Associates, Inc., Sales Brochure, Subject: *Advanced Continuous Simulation Language (ACSL)*, Concord, MA, May 1990.

2. COMSAT Laboratories, *ACS Simulator User's Manual*, Clarksburg, MD, March 1992.

3. The Aerospace Corporation, *PCSOAP, Personal Computer Satellite Orbit Analysis Program User's Guide Version 6.1.1.*, Computer Graphics Systems Department, El Segundo, CA, 1991.

4. Agrawal, B. N., *Design of Geosynchronous Spacecraft*, Prentice-Hall, Inc., 1986.

5. Kane, T. R., Likins, P. W., Levinson, D. A., *Spacecraft Dynamics*, McGraw-Hill, Inc., 1983.

6. Kolve, D. I., "Describing an Attitude", paper presented at the 16th Annual AAS Guidance and Control Conference, Keystone, CO, February 6-10, 1993.

7. Haynes, K. L., *Computer Graphics Tools for the Visualization of Spacecraft Dynamics*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1993.

8. Zyda, M. J., "Pixel Data to Image Display Tools to NPSGDL2", Part 5 of Class Notes for CS4202 - Computer Graphics, Naval Postgraduate School, Monterey, CA, May 1993

INITIAL DISTRIBUTION LIST

	Number of Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 052 Naval Postgraduate School Monterey, California 93943-5002	2
3. I. Michael Ross, Code AA/RO Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5002	6
4. Daniel J. Collins, Code AA Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5002	1
5. Robert B. McGhee, Code CS/MZ Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	1
6. David R. Pratt, Code CS/PR Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	1
7. Michael J. Zyda, Code CS/ZK Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	1
8. Randy L. Borchardt, Code EC/BT Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5002	1

9. LT Jeffrey A. Stewart 4
OC Division
USS Dwight D. Eisenhower (CVN-69)
FPO AE 09532-2830